Titre: Title:	Automatic Test Data Generation Using Constraint Programming and Search Based Software Engineering Techniques
Auteur: Author:	Abdelilah Sakti
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Sakti, A. (2014). Automatic Test Data Generation Using Constraint Programming and Search Based Software Engineering Techniques [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1655/

UNIVERSITÉ DE MONTRÉAL

AUTOMATIC TEST DATA GENERATION USING CONSTRAINT PROGRAMMING AND SEARCH BASED SOFTWARE ENGINEERING TECHNIQUES

ABDELILAH SAKTI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

AUTOMATIC TEST DATA GENERATION USING CONSTRAINT PROGRAMMING AND SEARCH BASED SOFTWARE ENGINEERING TECHNIQUES

présentée par: SAKTI Abdelilah
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. BELTRAME Giovanni, Ph.D., président
M. PESANT Gilles, Ph.D., membre et directeur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et codirecteur de recherche
M. ANTONIOL Giuliano, Ph.D., membre
M. HARMAN Mark, Ph.D., membre externe

This dissertation is dedicated to my wife, Khadija, my sons, Adam and Anas, and to my mother, Aïcha. Their support, encouragement, patience, and understanding have sustained me throughout my life...

ACKNOWLEDGMENTS

First of all, I would like to thank almighty ALLAH. Without his wish nothing is possible.

Completion of this doctoral research would not have been possible without the contributions of many people throughout the research project. Foremost are the generous support and patience of my supervisors.

I would like to take this opportunity to express my appreciation to my principal supervisor, Prof. Dr. Gilles Pesant, and associate supervisor Prof. Dr. Yann-Gaël Guéhéneuc, for their encouragement, advice, guidance, and inspiration throughout the duration of this research project. I have learned much from them about the attitudes and skills for conducting research and presenting ideas. They brought forth this research and allowed me to extend my education beyond the formal studies leading to this dissertation. They could not even realize how much I have learned from them. I am very obliged for their motivation and immense knowledge in Software Engineering that, taken together, make them great mentors.

I would also thank Prof. Dr. Giuliano Antoniol, for his feedback and productive discussions during the various stages of the research project.

I would also like to thank the members of my Ph.D. committee who enthusiastically accepted to monitor and read my dissertation.

I am very thankful to all my colleagues of Quosseca laboratory and Ptidej teams for their feedback and the productive discussions.

I am deeply grateful for the support my family provided during every stage of this dissertation.

Finally, appreciation is extended to the staff of the École Polytechnique de Montréal for their support throughout my research project.

RÉSUMÉ

Prouver qu'un logiciel correspond à sa spécification ou exposer des erreurs cachées dans son implémentation est une tâche de test très difficile, fastidieuse et peut coûter plus de 50% de coût total du logiciel. Durant la phase de test du logiciel, la génération des données de test est l'une des tâches les plus coûteuses. Par conséquent, l'automatisation de cette tâche permet de réduire considérablement le coût du logiciel, le temps de développement et les délais de commercialisation.

Plusieurs travaux de recherche ont proposé des approches automatisées pour générer des données de test. Certains de ces travaux ont montré que les techniques de génération des données de test qui sont basées sur des métaheuristiques (SB-STDG) peuvent générer automatiquement des données de test. Cependant, ces techniques sont très sensibles à leur orientation qui peut avoir un impact sur l'ensemble du processus de génération des données de test. Une insuffisance d'informations pertinentes sur le problème de génération des données de test peut affaiblir l'orientation et affecter négativement l'efficacité et l'effectivité de SB-STDG.

Dans cette thèse, notre proposition de recherche est d'analyser statiquement le code source pour identifier et extraire des informations pertinentes afin de les exploiter dans le processus de SB-STDG pourrait offrir davantage d'orientation et ainsi d'améliorer l'efficacité et l'effectivité de SB-STDG. Pour extraire des informations pertinentes pour l'orientation de SB-STDG, nous analysons de manière statique la structure interne du code source en se concentrant sur six caractéristiques, i.e., les constantes, les instructions conditionnelles, les arguments, les membres de données, les méthodes et les relations. En mettant l'accent sur ces caractéristiques et en utilisant différentes techniques existantes d'analyse statique, i.e, la programmation par contraintes (CP), la théorie du schéma et certains analyses statiques légères, nous proposons quatre approches : (1) en mettant l'accent sur les arguments et les instructions conditionnelles, nous définissons une approche hybride qui utilise les techniques de CP pour guider SB-STDG à réduire son espace de recherche ; (2) en mettant l'accent sur les instructions conditionnelles et en utilisant des techniques de CP, nous définissons deux nouvelles métriques qui mesurent la difficulté à satisfaire une branche (i.e., condition), d'où nous tirons deux nouvelles fonctions objectif pour guider SB-STDG ; (3) en mettant l'accent sur les instructions conditionnelles et en utilisant la théorie du schéma, nous adaptons l'algorithme génétique pour mieux répondre au problème de la génération de données de test ; (4) en mettant l'accent sur les arguments, les instructions conditionnelles, les constantes, les membres de données, les méthodes et les relations, et en utilisant des analyses statiques

légères, nous définissons un générateur d'instance qui génère des données de test candidates pertinentes et une nouvelle représentation du problème de génération des données de test orienté-objet qui réduit implicitement l'espace de recherche de SB-STDG.

Nous montrons que les analyses statiques aident à améliorer l'efficacité et l'effectivité de SB-STDG. Les résultats obtenus dans cette thèse montrent des améliorations importantes en termes d'efficacité et d'effectivité. Ils sont prometteurs et nous espérons que d'autres recherches dans le domaine de la génération des données de test pourraient améliorer davantage l'efficacité ou l'effectivité.

ABSTRACT

Proving that some software system corresponds to its specification or revealing hidden errors in its implementation is a time consuming and tedious testing process, accounting for 50% of the total software. Test-data generation is one of the most expensive parts of the software testing phase. Therefore, automating this task can significantly reduce software cost, development time, and time to market.

Many researchers have proposed automated approaches to generate test data. Among the proposed approaches, the literature showed that Search-Based Software Test-data Generation (SB-STDG) techniques can automatically generate test data. However, these techniques are very sensitive to their guidance which impact the whole test-data generation process. The insufficiency of information relevant about the test-data generation problem can weaken the SB-STDG guidance and negatively affect its efficiency and effectiveness.

In this dissertation, our thesis is statically analyzing source code to identify and extract relevant information to exploit them in the SB-STDG process could offer more guidance and thus improve the efficiency and effectiveness of SB-STDG. To extract information relevant for SB-STDG guidance, we statically analyze the internal structure of the source code focusing on six features, i.e., constants, conditional statements, arguments, data members, methods, and relationships. Focusing on these features and using different existing techniques of static analysis, i.e., constraints programming (CP), schema theory, and some lightweight static analyses, we propose four approaches: (1) focusing on arguments and conditional statements, we define a hybrid approach that uses CP techniques to guide SB-STDG in reducing its search space; (2) focusing on conditional statements and using CP techniques, we define two new metrics that measure the difficulty to satisfy a branch, hence we derive two new fitness functions to guide SB-STDG; (3) focusing on conditional statements and using schema theory, we tailor genetic algorithm to better fit the problem of test-data generation; (4) focusing on arguments, conditional statements, constants, data members, methods, and relationships, and using lightweight static analyses, we define an instance generator that generates relevant test-data candidates and a new representation of the problem of object-oriented test-data generation that implicitly reduces the SB-STDG search space.

We show that using static analyses improve the SB-STDG efficiency and effectiveness. The achieved results in this dissertation show an important improvements in terms of effectiveness and efficiency. They are promising and we hope that further research in the field of test-data generation might improve efficiency or effectiveness.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVATIONS

ATDG	Automatic Test Data Generation
CB-FF	Constraint Based Fitness Function
CB-STDG	Constraint Based Software Test Data Generation
CEO	Constrained Evolution Operator
CFG	Control Flow Graph
CUT	Class Under Test
CP	Constraint Programming
CPA-STDG	Constraint Programming Approach for Software Test Data Generation
CPG	Constrained Population Generator
CSB-STDG	Constrained-Search Based Software Test Data Generation
CSP	Constraint Satisfaction Problem
DFA	Data Flow Analysis
f_{DL}	Difficulty Level Fitness Functions
DSE	Dynamic Symbolic Execution
EGAF-STDG	Enhanced Genetic Algorithm Framework for Software Test Data Generation
GA	Genetic Algorithm
GOA	Goal Oriented Approach
IG-PR-IOOCC	Instance Generator and Problem Representation to Improve Object Oriented Code Coverage
IT	Information Technology
JPF	Java PathFinder
OOP	Object-Oriented Programming
POA	Path Oriented Approach
SB-STDG	Search Based Software Test Data Generation
SE	Symbolic Execution
SPF	Symbolic PathFinder
f_{SE}	Symbolically Enhanced Fitness Function
SSA-Form	Static Single Assignment Form
UUT	Unit Under Test

CHAPTER 1

INTRODUCTION

In the field of Information Technology (IT), software testing exists since the creation of the first computer system. In parallel to the digital revolution, software testing becomes mandatory because of the increasing demands and needs in computer systems that must behave systematically and that users can trust. Indeed, the consequences of poor software testing may be dire: software crashes and/or security breaches, which, in critical systems, may lead to financial losses or loss of life.

Critical systems are embedded in most of our daily activities. They are in transportation (e.g., avionics, rail, sea), in energy production and control systems (e.g., nuclear power centers, oil extraction), and also in the medical field (computer-aided surgery, automated medical treatments).

Poorly-tested critical systems have negatively affected our economy. Tassey (2002) reports that, inadequate software infrastructure costs the U.S. economy about $60 billion per year. Poorly-tested critical systems are at the root of the 1996 Ariane 5 incident during which an undesirable behavior caused by a floating-point conversion error led to the rocket self-destructing just 40 seconds after launching. The European Space Agency announced a delay in the project, estimated a direct economic loss of 500 million U.S. dollars and, implicitly, a loss of customers' confidence to the benefits of competitors (Garfinkel, 2005).

The disasters caused by poorly-tested critical systems motivated all stakeholders to invest in software testing. Software testing is a time consuming and tedious process, accounting for more than 50% of the total software cost (Pressman, 1992; Myers, 1979) and any technique reducing testing costs is likely to reduce the software development costs as a whole. Automatic Test Data Generation (ATDG) techniques can significantly reduce software cost, development time, and time to market (Ince, 1987). However generating a test-data set for an exhaustive and thorough testing is often infeasible because of the possibly infinite execution space and its high costs with respect to tight budget limitations. Techniques exist to select test data in order to reduce testing cost without compromising the software quality. Such techniques are recommended by several international standard (e.g., DO-178B for aircrafts, IEC 61513 for nuclear, IEC 50126 for railway.). For example, in the standard DO-178B (RTCA, 1992), programs are classified into five levels (from E to A), each level representing the possible effect of a program's error on passengers' safety. Level "A" has a catastrophic effect, Level "B" has a dangerous effect, Level "C" has a major effect, Level "D" has a minor effect, and Level "E"

has no effect on safety. The criterion for testing a program is defined by its classification in the standard (RTCA, 1992; Kong et Yan, 2010): for example, a C-level program must be tested by a test-data set that at least executes every program instruction once.

Black-box testing and white-box testing are the most dominant techniques in the field of software testing. Black-box testing is concerned with validating the software functionalities disregarding its implementation, whereas white-box testing is concerned only with the implementation of a software system (Myers, 1979).

1.1 Problem and Motivation

In this dissertation, we focus on *white-box testing*, also known as *structural testing*. Structural testing is commonly used with *unit testing* wherein each unit is tested separately (e.g., a unit is a procedure or function in a procedural programming language and a class in an object-oriented programming language). A requirement of structural testing is expressed in terms of the internal features of the Unit Under Test (UUT), i.e., an implementation (Vincenzi *et al.*, 2010): generally, a UUT is represented by a Control Flow Graph (CFG), then a test requirement is expressed in terms of CFG nodes and/or edges. For example, if we consider the function `foo` in Figure 1.1 as UUT, then a test requirement may be a test-data set that can reach all edges in Figure 1.2, i.e., all-branch coverage.

The literature describes automated techniques and tools to satisfy a test requirement by generating a test-data suite. Over the last two decades, researchers have focused mainly on Search Based Software Test Data Generation (SB-STDG) or Constraint Based Software Test Data Generation (CB-STDG) to generate test-data because these two approaches can achieve high coverage.

CB-STDG is a static approach: to generate test data, CB-STDG **statically** translates the test-data generation problem into a Constraint Satisfaction Problem (CSP), then uses a solver, i.e., a tool that takes a CSP as input and either computes a solution or proves its infeasibility. In the first case, the solution represents the test data needed to satisfy the test requirement; in the second case, the test requirement is unreachable. SB-STDG is a dynamic approach: to generate test data, SB-STDG starts by generating a random set of test-data candidates (e.g., an initial population), then for each candidate test-datum, an instrumented version of the UUT is executed and conditional statements are **dynamically** analyzed to determine a fitness value, i.e., a score that estimates how far a test-datum candidate is from the test target. Based on the fitness value and the search algorithm used, the current test-data candidates are evolved to generate a new set of test-data candidates and the approach continues, generating new sets of test-data candidates, until the test target is achieved or a

```
1  int foo(int x, int y, int z){
2      int maxHash=hash(x);
3      boolean b=(maxHash<hash(y));
4      if(b){
5        maxHash=hash(y);
6        b=(maxHash<hash(z));
7        if(b)
8          maxHash=hash(z);
9      }
10     else {
11         b=(maxHash<hash(z));
12         if(b)
13             maxHash=hash(z);
14      }
15     return maxHash;
16 }
```

Figure 1.1: The `foo` function

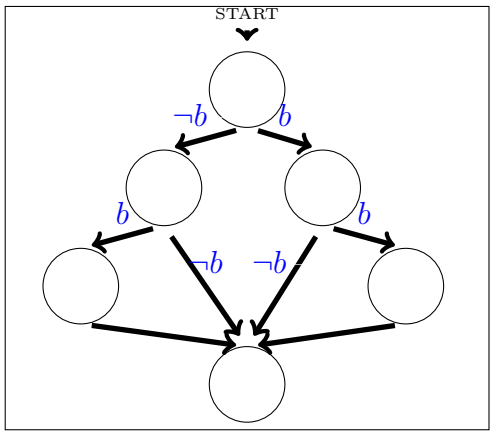

Figure 1.2: Control flow graph of the `foo` function

stopping criterion is reached.

Over the last decade, CB-STDG and SB-STDG approaches have been extensively explored (Collavizza *et al.*, 2010; Gotlieb, 2009; Lakhotia *et al.*, 2010; Malburg et Fraser, 2011; Păsăreanu et Rungta, 2010; Staats et Păsăreanu, 2010; Tillmann et de Halleux, 2008; Fraser et Zeller, 2012). The main advantages of the CB-STDG approach are in its precision in test-data generation and in its ability to prove that some execution paths are unreachable. Yet, scalability issue is the major disadvantage of this approach, i.e., it cannot manage the dynamic aspects of a UUT, e.g., dynamic structures, native function calls, and communication with the external environment (Tillmann et de Halleux, 2008). Thus, using CB-STDG, it is not always possible to generate an exact test data due to source code complexity or unavailability. For example, CB-STDG cannot generate test data that covers an execution path containing a native function call, e.g., any execution path in the function `foo` is problematic due to the native function call, `hash`, at Line 2 in Figure 1.1.

In contrast, SB-STDG approaches are scalable, i.e., they can deal with any sort of programs. But the efficiency and effectiveness of these approaches is sensitive to many factors: the search-space size; the fitness function; and the search heuristic. It is inefficient to use SB-STDG in a large search space without an efficient search heuristic and sufficient guidance. For example, to generate a test datum to execute a path involving a branch over a boolean, the fitness functions proposed in the literature may lack information to distinguish between test-data candidates (McMinn, 2004), e.g., in Figure 1.1, generating test data to reach the statement at Line 8 is difficult because the execution of the path leading to this line involves two conditional statements over boolean variables, i.e., at Lines 4 and 7.

CB-STDG derives its advantages from its static analyses that give it a global view on the whole test-data generation problem, whereas SB-STDG derives its advantages from its dynamic analyses that give it a complete view on a part of the test-data generation problem (i.e., a part of the execution path leading to a test target) and allows it to deal with any types of instructions. These different ways of analysis are also at the root of the disadvantages of each approach: because CB-STDG uses static analyses, it cannot analyze some sort of dynamic instructions (e.g., native function call) and because SB-STDG uses dynamic analyses, it has only a partial view on the problem, i.e., only the executed part of the test-data generation problem (e.g., execution path leading to a test target) is visible. Therefore, these approaches use complementary analyses that give them a potential for combination.

Because SB-STDG is scalable, it is used more often than CB-STDG to generate test-data in the software development industry (Parasoft, 2013). However, SB-STDG suffers from problems related to search guidance, e.g., the fitness function and the search heuristic (McMinn, 2004). These problems are less serious than the scalability issue and can be dealt with by offering relevant information about the global view of a test-data generation problem. Such information can be derived either directly, using static analyses, or indirectly, using CB-STDG.

1.2 Thesis

In our previous work (Sakti *et al.*, 2011; Bhattacharya *et al.*, 2011), we reviewed and analyzed the problem of test-data generation to understand the standard analyses used with the approaches SB-STDG and CB-STDG. We observed that there exist some internal features of UUT that may influence the process of test-data generation and statically analyzing them may lead to relevant information for the SB-STDG guidance. We identified six UUT features, i.e., arguments, conditional-statements (i.e., branch-conditions), constants, data members, methods, and relationship with other unite (e.g., inheritance, association, aggregation, or method call).

Many other researchers (Alshraideh et Bottaci, 2006; Harman *et al.*, 2007; Ribeiro *et al.*, 2008; McMinn *et al.*, 2010; Baars *et al.*, 2011; Zhang *et al.*, 2011; Alshahwan et Harman, 2011; Fraser et Zeller, 2011; McMinn *et al.*, 2012a; Fraser et Arcuri, 2012; McMinn *et al.*, 2012b) also discussed the importance of static analyses for SB-STDG. Table 1.1 summaries UUT features analyzed and programming language targeted in previous work: Arguments (Ar.), Conditional Statements (C.S.), Constants (Co.), Data Members (D.M.), Methods (Me.), relationship with other units (Re.), Object-Oriented language (O.O.), and Procedural Programming (P.P.).

Table 1.1: UUT features analyzed and programming language targeted in previous work.

Research work	Ar.	C.S.	Co.	D.M.	Me.	Re.	O.O.	P.P.
Harman *et al.* (2007)	√							√
McMinn *et al.* (2012a)	√							√
Ribeiro *et al.* (2008)					√		√	
Zhang *et al.* (2011)					√		√	
Baars *et al.* (2011)		√						√
Alshraideh et Bottaci (2006)			√				√	√
McMinn *et al.* (2010)			√				√	√
Alshahwan et Harman (2011)			√				√	√
Fraser et Zeller (2011)			√				√	√
Fraser et Arcuri (2012)			√				√	√
McMinn *et al.* (2012b)			√				√	√

Static analyses proposed in previous work does not consider all six features of a UUT and does not explore static analysis techniques such as constraint programming and schema theory. In addition, to the best of our knowledge, statically analyzing source code to extract and exploit relevant information about UUT features to guide SB-STDG in the field of object-oriented testing has not been analyzed yet except for constants. Thus, our thesis is:

> Statically analyzing source code to identify and extract relevant information to exploit them in the SB-STDG process either directly or through CB-STDG could offer more guidance and thus improve the efficiency and effectiveness of SB-STDG for object-oriented testing.

To prove our thesis, we define and adapt different static-analysis techniques to extract and exploit relevant information about each UUT feature: (1) focusing on arguments of a UUT, we define a CB-STDG approach and use it as static-analysis to guide SB-STDG in reducing its search space; (2) focusing on conditional-statements (i.e., branch-conditions) and using CP techniques, we define new fitness functions to guide SB-STDG; (3) focusing on conditional-statements (i.e., branch-conditions) and using schema theory, we define a new SB-STDG framework; (4) focusing on arguments, constants, data members, methods, and relationship with other units, we define a new static analysis techniques for object oriented test-data generation.

1.3 Contributions

The main contributions of this thesis are focusing on the use of static analyses to identify information relevant to guide SB-STDG search, extract them from the source code, and exploit them in the test-data generation process. First, we review and analyze the two widely used approaches in the field of automatic test-data generation, i.e., CB-STDG and SB-STDG, to identify UUT features that may influence the test-data generation process. Second, we enhance CB-STDG before using it as static analysis to guide SB-STDG. Finally, we propose many complementary static analyses based on different techniques: (1) a static analysis based on CP techniques (i.e., constraint arity and projection tightness (Pesant, 2005)) to improve fitness functions used with CB-STDG; (2) a static analysis based on schema theory (Holland, 1975) to improve the CB-STDG process, in particularly to tailor GA to better fit the problem of test-data generation; (3) many static analyses that focus on different UUT features (i.e., constants, data members, methods, branches, relationship with other unites) to offer relevant information for SB-STDG about Object-Oriented Programming (OOP).

More specifically, the main contributions are:

1. CPA-STDG, a new CB-STDG approach whose goal is exploiting CP heuristics to mitigate CB-STDG deficiencies (e.g., the exploration of execution paths faces the combinatorial explosion problem). CPA-STDG explores execution paths by using CP heuristics. It uses CFG to model all execution paths and link them to constraints representing the program statements. CPA-STDG has two main advantages: (1) exploring execution paths in a solver can benefit from CP heuristics and avoid exploring a significant number of infeasible paths and (2) modeling a CFG keeps the program semantics and facilitates identifying its internal structure, thus it could easily answer to different test requirements. The results of this contribution were published in Actes des 7e Journées Francophones de Programmation par Contraintes (JFPC) (Sakti *et al.*, 2011) and the 3rd IEEE International Symposium on Search-based Software Engineering (SSBSE) (Bhattacharya *et al.*, 2011).

2. CSB-STDG, a new approach that combines CPA-STDG and SB-STDG to achieve better code coverage. It simplifies the problem of test-data generation to solve it using CPA-STDG and uses the solutions as a reduced search space with SB-STDG. The results of this contribution were published in the 4th IEEE International Symposium on Search Based Software Engineering (SSBSE) (Sakti *et al.*, 2012).

3. CB-FF, a novel fitness functions based on branch "hardness" whose goal is using CP techniques to better guide a SB-STDG approach, thus reach higher code coverage. It

statically analyzes the test target to collect information about branches leading to it. Then it defines a penalty value for each branch according to CP techniques, i.e., its arity and its constrainedness. Branch penalties are used to determine how close a test-data candidate is to reach a test target. The results of this contribution were published in the 10th International Conference on Integration of Artificial Intelligence and Operations Research in Constraint Programming (CPAIOR) (Sakti *et al.*, 2013).

4. Enhanced Genetic Algorithm Framework for Software Test Data Generation (EGAF-STDG), a new SB-STDG approach that tailors GA to better match the problem of test-data generation, thus reaching better coverage. It uses schema theory (Holland, 1975) to analyze and restructure the problem and identify properties associated with better performance, then it incorporates them in the evolution phase with all fundamental GA operators (i.e., selection, crossover, and mutation). EGAF-STDG increases the code coverage achieved by a simple GA by restructuring the problem of test-data generation in a new representation wherein GA has a high likelihood to succeed.

5. Instance Generator and Problem Representation to Improve Object Oriented Code Coverage (IG-PR-IOOCC), a new SB-STDG approach for unit-class testing that reduces the search space by analyzing statically the internal structure of a Class Under Test (CUT), generating relevant instances of classes using diversification and seeding strategies, and thus achieving better code coverage. The results of this contribution were published in IEEE Transactions on Software Engineering (TSE) (Sakti *et al.*, 2014).

1.4 Organization of the Dissertation

The rest of this dissertation is organized as follows.

Chapter 2 AUTOMATIC TEST DATA GENERATION presents the techniques and concepts that we use in this dissertation. The chapter starts by briefly describing the test-data generation process. It continues by introducing the principles of SB-STDG and CB-STDG.

Chapter 3 RELATED WORK presents works and research areas that are related to our dissertation. The chapter starts by briefly presenting the state of the arts of CB-STDG and SB-STDG. Next, it presents the state of the art of combining CB-STDG and SB-STDG. It finishes with the state of the art of test-data generation approaches for object-oriented programming.

Chapter 4 CP APPROACH FOR SOFTWARE TEST DATA GENERATION starts by describing the methodology that we propose to translate a test-data generation problem into a constraints satisfaction problem. It continues with designing the modeling constraints for control-flow criteria and data-flow criteria. It provides an empirical study to compare our approach with two CB-STDG approaches from the literature. Then, it concludes with a discussion of the results.

Chapter 5 COMBINING CB-STDG AND SB-STDG This chapter presents a hybrid test-data generation approach that combines CB-STDG and SB-STDG. The chapter starts by explaining the limitations of both approaches and shows the potential of their combination. It continues by presenting different techniques of combination. Then, it provides an empirical experimental comparison of the proposed approach with a CB-STDG from the literature. The chapter concludes with a discussion of the results.

Chapter 6 CONSTRAINT BASED FITNESS FUNCTION presents an approach that exploits constraint programming techniques to offer a new fitness function for SB-STDG. The chapter starts by presenting the limitations of existing fitness functions. It continues with presenting metrics to evaluate a test-data candidates. It provides an empirical experimental comparison of the proposed fitness functions and the state of the art.

Chapter 7 SCHEMA THEORY OF GENETIC ALGORITHMS TO ANALYZE AND IMPROVE SB-STDG presents the concept of schema theory and uses it to enhance evolutionary testing. It explains how to use information from schema theory in different search stages to get better performance. Also, it provides the details of an empirical experimental study that compares the proposed approach to an existing evolutionary testing approach.

Chapter 8 FURTHER STATIC ANALYSES TO IMPROVE SB-STDG presents a new approach that exploits static analysis to enhance SB-STDG for OOP. It continues presenting different weak points in the process of SB-STDG for Object-Oriented programming and how to deal with them. It continues by describing an implementation, JTExpert, of the approach. Also, it provides an empirical experimental comparison of the proposed approach and the state of the art. The chapter concludes based on the results and discussion of experiments.

Chapter 9 JTEXPERT: A LARGE SCALE EXPERIMENTATION presents an adaptation of our approach IG-PR-IOOCC and its implementation JTExpert to raise excep-

tions via branch coverage, i.e., an Exceptions-oriented Test-data Generation Approach. It continues by presenting experimental results of applying JTExpert on a large open-source software, the Apache Hadoop System.

Chapter 10 CONCLUSION AND FUTURE WORK revisits the main thesis and contributions of this dissertation. It also describes potential opportunities for future research.

Part I

Background

CHAPTER 2

AUTOMATIC TEST DATA GENERATION

This chapter provides the details of the three main techniques, i.e., ATDG, SB-STDG, and CB-STDG which we used in this dissertation. First, we briefly explain the ATDG process for structural testing. Second, we explain the principle of SB-STDG and the main fitness functions and search heuristics used in this approaches, i.e., random search, GAs, branch distance, and approximation level which we enhance or compare with. Third, we explain the principle of CB-STDG and the main techniques, i.e., POA and GOA we improve and compare with.

Software testing is a main phase of the software development life cycle. It aims to find faults or gain confidence in a software system. This phase requires an executable software called *software under test*, an input data called *test data* or *test input*, and an expected output called *oracle* (Machado *et al.*, 2010). The combination of a test data and its oracle is called *a test case*. Software testing consists of generating test data, generating an oracle for each test data, and exercising the software under test by using the test data and comparing its outputs to the oracle. The large or infinite input and output domains may make exhaustive testing impossible, i.e., testing a software using all the possible inputs. To overcome this challenge and make testing scalable without compromising the testing quality, Goodenough et Gerhart (1975) defined a theory of test-data selection and the notion of test criteria. Since then, test criteria have been a major research focus in the field of software testing. Consequentially, a wide set of test criteria have been proposed, investigated, and classified into two testing strategies, i.e., black-box testing and white-box testing (Myers, 1979). *Black-box testing* is concerned with validating the software functionalities disregarding its implementation, whereas *white-box testing* is concerned only with the implementation of a software system (Myers, 1979). These two testing strategies are complementary, i.e., a missing functionality cannot be revealed by white-box testing, unexpected functionality cannot be detected by black-box testing. In this dissertation we are interested only in white-box testing.

2.1 White-box Testing

White-box Testing, also known as *structural testing*, is commonly used with *unit testing* where each *unit* is tested separately (e.g., a unit is a procedure or a function in a procedural programming language and it is a class in a object-oriented programming language). To

determine the testing requirements, the internal features of a unit are used, i.e, implementation (Vincenzi *et al.*, 2010): a unit is represented as a CFG and a test requirement called a *coverage criterion* is expressed in terms of nodes and edges, e.g., it may be a subset of edges, a subset of nodes, or a combination of thereof. Generally, to formally define a requirement of white-box testing two items are used: a CFG and a coverage criteria.

2.1.1 Control Flow Graph (CFG)

CFG of a program is a directed graph whose nodes represents a sequence of instructions, which together constitute a *basic block*, and edges represent possible transitions between nodes (Ferrante *et al.*, 1987). A CFG has a single input node called $START$ and one output node called END. For each node N of a CFG, there exists at least one path from $START$ to N and another from N to END. A *complete execution path* is a path from START to END. Any program can be represented by a control flow graph. Figure 2.2 represents a CFG of the function `triangle` shown in Figure 2.1. Such a structure is used to express different structural test-criteria that are known as coverage criteria.

2.1.2 Coverage Criterion

A coverage criterion or, as originally defined by Goodenough et Gerhart (1975), selection criterion, is a predicate that defines properties of a program that must be exercised. Formally, it defines the set of CFG paths that must be traversed to test a program.

Structural coverage criteria divide in two categories: data-flow based and control-flow based. Well-known control-flow based structural coverage criteria are all-nodes (known also as all-statements), all-edges (known also as all-branches), or all-paths. Data-flow based coverage criteria focuses on the interactions among variables and their effect on program statements. A data-flow coverage criterion is expressed in terms of variable definitions and their usages (e.g., all-def, all-use). To express a data-flow criterion over CFG nodes a Data Flow Analysis (DFA) is required.

2.1.3 Data Flow Analysis

During program execution, a program instruction handles variables in two ways: *def* when the variable value is changed, the CFG node that represents this statement is a *def-node* of that variable and *use* when the variable value is used in an expression (e.g., referred in right side of an assignment statement), the CFG node that represents this statement is a *use-node* of that variable. A def or a use can be identified by a pair $\langle v, N_i \rangle$ where v is a program variable and N_i is a CFG node. A use is split into two types: *c-use* if the use-node is a

```
  int NOT_A_TRIANGLE=0;
  int SCALENE=1;
  int EQUILATERAL=2;
  int ISOSCELES=3;
01 int Triangle(int a, int b, int c)
02 {
03    int type;
04    int t;
05    if (a > b){
06-08         t = a; a = b; b = t;
09    }
10    if (a > c){
11-13         t = a; a = c; c = t;
14    }
15    if (b > c){
16-18         t = b; b = c; c = t;
19    }
20    if (a + b <= c){
21          type = NOT_A_TRIANGLE;
22    }
23    else{
24         type = SCALENE;
25         if (a == b && b == c){
26              type = EQUILATERAL;
27         }
28         else {
29              if (a == b || b == c){
30                   type = ISOSCELES;
31              }
32         }
33    }
34    return type;
35 }
```

Figure 2.1: The `triangle` function.

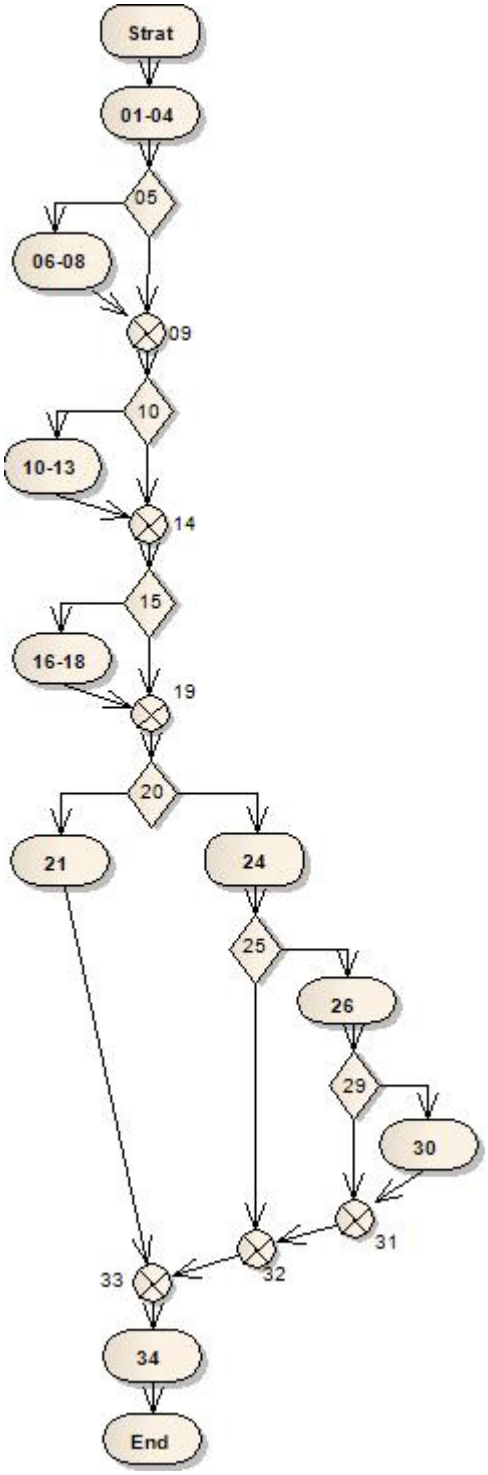

Figure 2.2: The CFG corresponding to `triangle` function.

statement-node; *p-use* predicate use if the use-node is a decision-node. A triplet $\langle v, N_d, N_u \rangle$ is a *def-use association* if there exist a *def-clear* path between N_d and N_u with respect to the variable v. A path is *def-clear* if it does not contains any new definition of that variable and its first node N_d is a def-node of the variable v and its last node N_u is an use-node of that variable. A node that prevents a path to be def-clear for a def-use $\langle v, N_d, N_u \rangle$ is a *killer-node*, it is a def-node of v, located between N_d and N_u, that replaces the currently active definition of v with its own definition. *Data flow analysis* is the process that determines all def-use associations.

Example: coverage criterion The `triangle` program is a well benchmark (Bardin *et al.*, 2009; Botella *et al.*, 2002; Williams *et al.*, 2005). It allows users to identify a triangle by the relative length of its three sides. The code in Figure 2.1 is an implementation of this program (McMinn, 2004). The `triangle` function takes as input three integers and returns 0 if these cannot be the lengths of the sides of a triangle; 1 if the parameters come from a scalene triangle; 2 from an equilateral triangle; 3 from an isosceles triangle.

We suppose that the unit under test is the function `Triangle` in Figure 2.1 and the coverage criterion requires to traverse all edges in Figure 2.2, i.e., all-edge or all-branch coverage criterion. The requirement of this criterion is the set of all edges $E = \{\langle 05, 06-08 \rangle, \langle 05, 09 \rangle, \langle 10, 11-13 \rangle, \langle 10, 14 \rangle, \langle 15, 16-18 \rangle, \langle 15, 11-19 \rangle, \langle 20, 21 \rangle, \langle 20, 24 \rangle, \langle 25, 24 \rangle, \langle 25, 32 \rangle, \langle 29, 30 \rangle, \langle 29, 31 \rangle\}$ any edge in the CFG and not in `E` (e.g. $\langle 14, 15 \rangle$) it implicitly covered. According to this requirement, the test-data generation task consists of finding a set of test input `T` that covers all-branch. Formally, `T` is *All-branches-adequate* if and only if $\forall e \in E, \exists t \in T / triangle(a_t, b_t, c_t)$ traverse e. In the example the set $T = \{\langle 1, 2, 3 \rangle, \langle 6, 5, 4 \rangle, \langle 6, 6, 6 \rangle, \langle 4, 4, 6 \rangle\}$ is All-branches-adequate because the four executions of the function `triangle` by using the four 3-tuplet in `T` exercise all edges.

On this small example the test-data generation task is simple. However, this task is very difficult for a real program. In general, generating test data manually is unproductive and error-prone (Machado *et al.*, 2010). Therefore, automating this testing task is crucial to achieve a high level of code coverage. To make possible automating test-data generation, many approaches have been developed since 1976 when the first automated test-data generation approaches appeared (Miller et Spooner, 1976; Clarke, 1976). The two widely used approaches are SB-STDG and CB-STDG.

2.2 Search Based Software Test Data Generation (SB-STDG)

SB-STDG was introduced by Miller et Spooner (1976). To generate test inputs, SB-STDG uses a search algorithm that is guided by an objective function or fitness function. The fitness

function is defined according to a desirable test target. The commonly used fitness functions are branch-distance and approach-level (McMinn, 2004). To generate test data, SB-STDG starts by generating a random set of test-data candidates (e.g., an initial population). For each test-datum candidate, an instrumented version of the Unit Under Test (UUT) is executed and its fitness is computed. Based on the fitness value and the search algorithm used, the current test-data candidates are evolved to generate a new set of test-data candidates. acSB-STDG continues generating new set of test candidates until the test target is achieved or a stopping criterion is reached.

2.2.1 Fitness Function

The fitness function assigns a score to each test-datum candidate by measuring how far a test-datum candidate is from the test target. A dynamic family of fitness functions is used in the literature. This family assigns a score to a test-datum candidate i by executing an instrumented version of the UUT using i as input vector.

The *approximation-level* (Wegener *et al.*, 2001) and *branch-distance* (Tracey *et al.*, 1998) measures are used to define fitness functions. These measures are respectively the number of branches leading to the test target that were not executed by the test candidate and the minimum required distance to satisfy the branch condition where the execution of the test candidate has left the target path, e.g., the branch distance to satisfy the true branch of a conditional statement at Line 25 in Figure 2.1 `if(a==b && b==c)` is equal to $|a-b|+|b-c|$. The values of a, b, and c are extracted during the execution of Line 25. In general, the branch distance is used after a normalization, i.e., the branch distance is converted to a number in the interval $[0, 1]$ using either of the following formulations (Arcuri, 2010): $\eta(\delta) = 1 - 1.001^{-\delta}$ or $\eta(\delta) = \frac{\delta}{\delta+1}$ where δ is the branch distance.

For better accuracy, a new hybrid family of fitness functions has been proposed in the literature: Baars *et al.* (2011) propose Symbolically Enhanced Fitness Function (f_{SE}) and in a previous work we propose Difficulty Level Fitness Functions (f_{DL}, f_{DC}) (Sakti *et al.*, 2013). In addition to the dynamic computation, this family uses static analyses that take in consideration the non-executed branches. In addition to the score realized at the critical branch, this family adds a score for each non-executed branch on which the test target is control dependent.

The fitness function with the search heuristic are keys to the success of a SB-STDG technique. This couple represent the heart of SB-STDG that may distinguish between two SB-STDG techniques. In this dissertation, we use many fitness functions and mainly two search heuristics: random search and Genetic Algorithm (GA).

2.2.2 Random Search

Random search is the simplest among the search heuristics, it is largely used in software testing because it may lead to a high level of coverage. The common random search relies on generating a test-data candidate vector (i.e., a value for each input variable) randomly to execute the instrumented unit and reach the targeted branch. It stops either if the generated test candidate vector executes the targeted branch or if a stop condition is reached. For example, in the `triangle` example to reach the branch $\langle 29, 30\rangle$ or the statement at Line 30 in Figure 2.2 a random search randomly generates three integer values $\langle a_v, b_v, c_v\rangle$ to execute an instrumented version of the function `triangle`, i.e., a version that contains some additional statements for tracking the execution and variable values. If the statement at Line 30 is executed then the search stops and a test input is returned otherwise the search restarts a new iteration until a stopping criterion is met.

The main advantages of random search is that it is cheap, simple, and easy to implement and it may reach a high level of code coverage. However, it is a blind and shallow search that may not reach some important test targets if they are unlikely to be covered randomly. Thus, random search does not ensure stable strength of coverage. There exist other sophisticated search heuristics such as GA that keep advantages of random search while mitigating its disadvantages.

2.2.3 Genetic Algorithm (GA)

GA is a meta-heuristic search technique that is based on the idea of natural genetics and Darwin's theory of biological evolution (Holland, 1975; Goldberg, 1989; Mitchell, 1999). It supposes that new and fitter sets of individuals may be created by combining portions of fittest individuals in the previous population generation. A GA aims to reach an optimal solution of a problem by exploring potential solutions in its search space. A GA has typically six main components: (1) a representation of a feasible solution to the problem, called *chromosome*, each component of which is a *gene*; (2) a population of encoded individuals; (3) a fitness function to assign a score to each individual; (4) a selection operator to select parents according to their fitness; (5) a crossover operator to recombine and create new offspring; and, (6) a mutation operator to diversify the population by introducing new offspring.

The GA processes populations of individuals, successively replacing one such population with an evolved one. It works as follows:

1. Randomly generate a population of n individuals;

2. Evaluate the fitness $f(i)$ of each individual i in the population;

3. Exit if a stopping condition is fulfilled;
4. Repeat the following steps until n offspring have been created:
   (a) Select a pair of parents from the current population;
   (b) Recombine the two selected parents and create two new offspring;
   (c) Mutate the two offspring.
5. Replace the population by the new one and return to Step 2.

As with any evolutionary search method, the way in which individuals are evolved is the key factor in the success of a GA. The GA evolution phase involves three fundamental operators: selection, crossover, and mutation.

Selection

A selection operator (scheme) selects individuals in the population for reproduction. A fitter individual is likely to be selected to reproduce many times. In the literature, many selection operators have been proposed for evolutionary algorithms. An analysis of commonly used selection operators can be found in (Goldberg et Deb, 1991; Blickle et Thiele, 1996; Mitchell, 1999).

Crossover

A crossover operator recombines individuals via an exchange of genes between pairs of chromosomes. It recombines individuals with the hypothesis that the fittest individuals can produce even fitter ones. There are three commonly used crossover techniques: one-point, two-point, and uniform. Depending on the crossover technique and a predefined probability (i.e., *crossover rate*), the crossover randomly selects points of crossover (gene positions) and exchanges genes between two parents to create new offspring.

Mutation

A mutation operator is a process wherein a gene is randomly modified to produce a new chromosome. As with the crossover, the mutation occurs only with some probability (i.e., *mutation rate*). Any gene in a chromosome can be subject to mutation. A mutation operator prevents the GA to be trapped in a local optima (i.e., phenomena of premature convergence).

Schema Theory

Schema theory serves as the analysis tool for the GAs process to explain how and in which class of problem GAs work well (Holland, 1975). If the search space can be split into sub-spaces wherein chromosomes have some features in common, then GAs have the potential to work well to solve such a problem. In this class of problem, GAs are well suited because they works according to the divide-and-conquer principle but do not require that the sub-problems be independent. Thus, the problem is solved using problem decomposition and the assembly of the solution from sub-solutions (Watson *et al.*, 1998). A hard problem is reduced to a set of easier ones, solved more easily than the original one. Schema theory has been originally defined for binary strings representation to provide a formal framework for the informal *building-block* notion (Mitchell, 1999) but it can be adapted to any other representation.

A *schema* is a pattern representing a subset of the search space wherein chromosomes have some features in common (Holland, 1975). On binary strings, a schema is a template of chromosomes drawn from the set $\{0, 1, *\}$, where the asterisk represents a wild card. For example, a string *str* of one character coded by a chromosome of eight binary bits is an instance of the schema "$s = 0 * *1 * * *$" if and only if the *str* chromosome contains "0" at the position 1 and "1" at the position 4. This example of schema has an *order* equal to 2 (the number of fixed bits) and a *length* equal to 3 (the number of bits between the first and the last fixed bits).

The fitness value of a schema is equal to the average fitness value of its instances. Individuals from a schema having a fitness value higher than the average are likely to produce fitter individuals (Holland, 1975). Therefore, schema theory can be used to analyze the potential of using GAs with a given fitness function to solve the problem of test-data generation.

Evolutionary Testing

Evolutionary testing is a variant of SB-STDG that solves the test-data generation problem using an evolutionary algorithm, e.g., a GA.

As example, let us suppose that the test requirements is reaching Line 26 (requires sides of an equilateral triangle) in the function `triangle` defined in Figure 2.1. Let us apply the GA steps presented in Subsection 2.2.3. We denote the iteration number by i, initialized with value 0.

1. Generation of an initial population. Suppose that the size of the population is 5 and the following population was randomly generated.
   $P_i = \{ \langle 5, 3, 2\rangle, \langle 1, 1, 2\rangle, \langle 3, 3, 2\rangle, \langle 6, 4, 1\rangle, \langle 4, 4, 7\rangle\}$

2. Fitness evaluation. Assume that the fitness function used is approach-level (Wegener *et al.*, 2001) $f(t) = l_t + \eta(\delta_t)$
   $f(P_i) = \{ \langle l_1 = 2, \delta_1 = 1, f_1 = 2.50\rangle, \langle l_2 = 2, \delta_2 = 1, f_2 = 2.50\rangle, \langle l_3 = 1, \delta_3 = 1, f_3 = 1.50\rangle, \langle l_4 = 1, \delta_4 = 5, f_4 = 1.83\rangle, \langle l_5 = 1, \delta_5 = 3, f_5 = 1.75\rangle\}$
   Because the objective is a fitness value equal to 0, the best individual is the 5^{th} one, with the smallest fitness value.

3. Because no individual could reach Line 26, the search goes to the next steps that consist of evolving the best individuals.

4. Evolution.

   (a) The pair of individuals $\langle 5, 4\rangle$ is selected for evolution;

   (b) The recombination operator produced $\{\langle 6, 4, 7\rangle, \langle 4, 4, 1\rangle\}$, in which the third gene was swapped.

   (c) Let us suppose that the mutation operator produced $\{\langle 5, 4, 7\rangle, \langle 4, 4, 4\rangle\}$. The second individual is a solution;

   The two generated individuals are inserted in a new population P_{new}. The operations (a), (b), and (c) are repeated until the size of P_{new} reaches step 5.

5. P_{new} replaces P_i and a new iteration is started from 2. Because the new population contains a solution, $\langle 4, 4, 4\rangle$, the search is stopped at the second iteration.

In the example, the problem of test-data generation is encoded by a vector of three integer values where an individual is a direct representation of the input variables. Thus, an optimal solution is an individual that can reach the test target, i.e., which satisfies all control dependent branches of the test target.

The example is simple to encode and easy to solve because it represents the problem of test-data generation for procedural programming. Although, the encoding and the solving process may be more complex when considering the problem for OOP.

2.2.4 SB-STDG for Object Oriented Programming

Test-data generation for OOP is challenging because of the features of OOP, e.g., abstraction, encapsulation, and visibility that prevent direct access to some parts of the source code. When considering the problem of test-data generation for OOP, SB-STDG must take into account the state of the objects. Before invoking a method that may reach a test target, an instance of the CUT and a sequence of method calls are required to put the CUT in a desirable state to reach a test target.

To address the accessibility problem, a SB-STDG approach must performs three actions: (1) instantiate the CUT and all other required classes; (2) perform a sequence of method calls to put the instance of the CUT in a desired state (i.e., a state that may help to reach the test target); and, (3) call a method that may reach the test target. The problem is in finding an adequate combination of these three actions to represent a test data to reach a given test target. Solving this problem faces three difficulties: ($D1$) finding an adequate instance of the CUT and of each required object; ($D2$) finding an adequate sequence of method calls to put the instance of the CUT in a desired state; and, ($D3$) finding an adequate method to reach the test target.

Therefore, the search problem consists of finding a means to instantiate the CUT (D1), a sequence of method calls (D2+D3), and an instantiation for each required argument (D1). To address the difficulties of test-data generation for unit-class testing, the most widely used approach is exploring the whole search space of D1 and a reduced search space of D2+D3 that bounds the length of sequences of method calls (Tonella, 2004; Pargas *et al.*, 1999; Xie *et al.*, 2005; Arcuri et Yao, 2008; Wappler et Wegener, 2006; Fraser et Arcuri, 2011). Initially, such an approach randomly generates instances of classes and sequences of method calls. To generate a sequence of method calls, a length ℓ_r bounded by a constant L is randomly generated, where L is specified by the user or chosen automatically by the approach, then an algorithm iteratively and randomly selects an accessible method until the sequence length reaches ℓ_r. Instances of classes and sequences of method calls are evolved or regenerated until a stopping condition is reached.

2.3 Constraint Based Software Test Data Generation (CB-STDG)

CB-STDG was introduced in the 1970s (Clarke, 1976; Boyer *et al.*, 1975). At that time, limitations in hardware and decision algorithms prevented the development of CB-STDG. In 1993, in parallel with the evolution of hardware and decision procedures, the work of Demillo and Offut (DeMillo et Offutt, 1993) relaunched CB-STDG. During the two last decades several researchers have shown interest in CB-STDG (Gotlieb *et al.*, 2000; Collavizza et Rueher, 2006; Bardin *et al.*, 2009; Inkumsah et Xie, 2007; Visser *et al.*, 2004; Baars *et al.*, 2011; Malburg et Fraser, 2011).

CB-STDG is an approach of ATDG that consists of using CP techniques to solve the problem test-data generation.

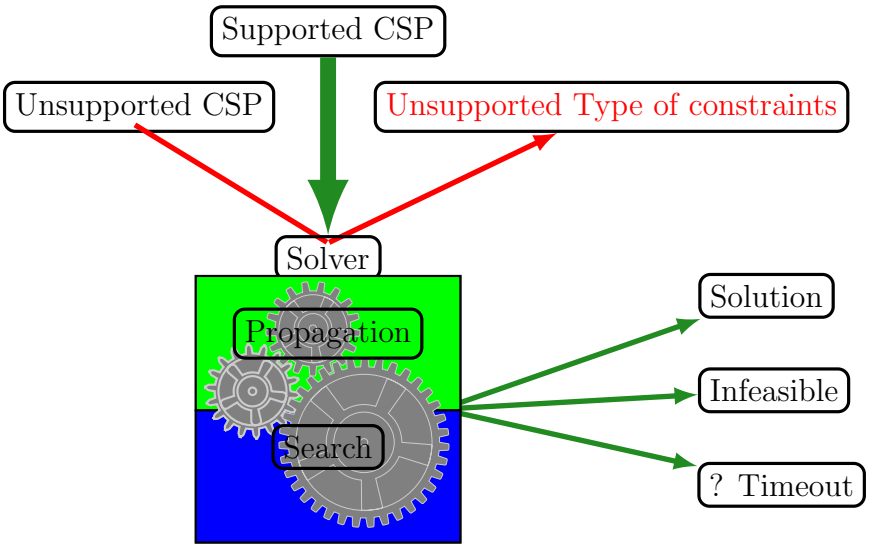

Figure 2.3: An abstraction of a solver

2.3.1 Constraint Programming (CP)

CP is a high-level framework that uses several techniques (filtering, propagation, search heuristics) to find solutions to combinatorial problems in a reasonable computational time. These problems can be classified as NP-complete or NP-hard. The power of CP is in its effectiveness in resolving problems, and its simplicity of use in different applications (Puget, 1995). CP is a technique to solve CSPs that uses logical inference within each constraint and propagates it among constraints. It is usually combined with tree search (problem decomposition) to make it deterministic.

A Constraint Satisfaction Problem (CSP) is a triplet (X, D, C) where X is a set of variables $\{x_1, ..., x_n\}$, D is a set of domains $\{D_1, ..., D_n\}$, and C is a set of relationships over the set of variables, called constraints. Each domain D_i is a set of possible values for each x_i. Each constraint expresses a property that must be satisfied by the values of the CSP variables. An assignment of values to the CSP variables is a solution if and only if it satisfies all constraints.

Although CP techniques have been designed for solving problems in logistics, production, and scheduling; during the last two decades, the applications of CP increased and its techniques have been adapted to effectively solve problems in others domains: hardware testing and verification (real-time system) (Adir et Naveh, 2011; Moss, 2008, 2010; NAVEH *et al.*, 2007) as well as software testing and verification (Gupta, 2008).

2.3.2 Solver

A solver is a tool that implements CP techniques. It takes a CSP as input and computes a solution or proves its infeasibility. Figure 2.3 shows an abstraction of a solver and the different answers that can be produced. When designing and implementing a solver, the aim is creating a library that can easily be reused to solve CSPs. A solver typically is composed of two layers: propagation and search. The propagation layer is based on mathematical theories to reduce the search space. It ensure the local consistency of each constraint, i.e., it treats each constraint separately and removes any inconsistent value from the domains of their variables. The search layer consists of fixing variables according to a certain strategy, e.g., the variable with the smallest domain is randomly fixed to one value from its domain. Each time a variable is fixed by the search layer, the propagation layer considers this change and uses constraints in which that variable is involved to remove all inconsistent values from the domains of other variables. After fixing a variable x to a value v, if the domain of any other variable becomes empty then a backtrack is done by returning into the state before fixing the variable x and the value v is removed from the domain of x. A solution is reached if a solver can fix all variables. A CSP is proved infeasible if the domain of a variable becomes empty and there is no backtrack possible.

2.3.3 CB-STDG principle

In general, to generate test inputs, CB-STDG translates the test-data generation problem into a CSP, which is either solved or proved infeasible; in the first case, the solution represents the test inputs needed to reach a given test target. A test-data generation problem may generate complex constraints over different data types (e.g., integer, float, string, arrays) and require a complex combination of theories that may be impossible to solve. Therefore, the CB-STDG works well as long as we can translate the whole testing problem into a CSP. It becomes problematic when CSP translation meets some complex instructions (e.g., recursive data structures, unbounded arrays, floating point numbers, different data types, random memory access through pointers) or if a part of the UUT source-code is inaccessible, dynamically generated, or a black box (e.g., native function calls, sensor interface). Therefore, the effectiveness and limits of a CB-STDG approach essentially depend on two main things: how the translation process generates the CSP and the solver used.

Several approaches have been proposed to translate the problem of test data generation into a CSP, then solve it using a solver. Well known approaches are the path oriented approach and the goal oriented approach. The main difference between these two approaches is in the way a test data generation problem is translated into a CSP.

2.3.4 Path Oriented Approach (POA)

The POA subdivides the test data generation problem into sub-problems, i.e., each path leading to the test target is a sub-problem. One sub-problem at time is solved until the test target is reached or a stopping condition is met. If the selected path is reachable then a test input is generated to exercise that path otherwise a different sub-problem is selected. POA builds sub-problems from a symbolic tree or by executing an instrumented version of a UUT and then by generating the path condition over symbolic input variables.

The purpose of the POA is to generate a test-datum set to observe all possible behaviors of a UUT, i.e., the objective is generating a test-data set from a symbolic execution tree, which is an abstract representation of all execution paths. In practice, it is impossible to calculate and solve constraints generated by all possible paths in a UUT (the number of paths is infinite with the presence of loops or recursion). In the existence of recursion or loops there exist a significant or infinite number of execution paths to explore. Some conventional approaches of exploring trees are used (e.g., depth-first), which may face the combinatorial explosion in some types of trees (Xie *et al.*, 2009). Therefore, execution paths exploration strategy is the main disadvantage of the POA. The straightforward solution to deal with this problem is by bounding the depth of a execution path or the number of iterations of a loop with a small constant K, where K is specified by the user or chosen automatically by the approach. The problem with this solution is that a reachable path may become an unreachable.

Symbolic Execution (SE)

SE is a POA approach that was introduced in the 1970s (Clarke, 1976; King, 1976). It is performed by executing a UUT using symbolic input variables instead of concrete input values. When symbolic execution meet a variable assignation, this variable is symbolically evaluated by replacing any variable in the assigned expression with its symbolic evaluation. Thus, an expression of a conditional statement is symbolically evaluated and each of its branches is assigned a symbolic expression. A program can be represented by a *symbolic execution tree* whose nodes are conditional statements and an edge is a possible branch labeled with its symbolic expression. The conjunction of edges' labels throughout an execution path in a symbolic execution tree forms a *path condition*. Solving a path condition generates test inputs needed to exercise the equivalent path in the UUT.

As an example, in Figure 2.5, on the left side, the function `foo` returns the maximum number among three integers. Let us suppose that the function `foo` is a UUT and the test target is reaching Line 6 ($max = z$). A representation of the symbolic execution tree of the function `foo` is given on the right side of Figure 2.5. To generate a test input that executes

```
1  int foo(int x, int y, int z){
2      int max=x;
3      if(max<y){
4        max=y;
5        if(max<z)
6          max=z;
7      }
8      else
9        if(max<z)
10         max=z;
11     return max;
12 }
```

Figure 2.4: Symbolic execution example: `foo` function is a UUT

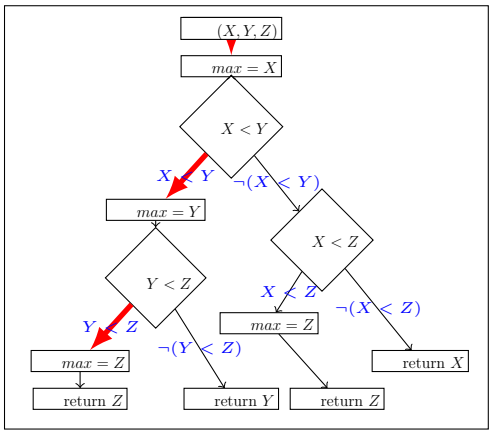

Figure 2.5: Symbolic tree of foo function

the statement at Line 6, a symbolic execution approach explores all paths that start at the root of the symbolic tee and end at the test target, i.e., paths going to Line 6. In this example, there exist only one path that can reach the test target, highlighted in red in the symbolic tree. Its constraint is $(X < Y) \wedge (Y < Z)$. The path condition is solved using a solver and the test inputs are generated. The triplet $(0, 1, 2)$ is a solution of that path condition and the execution of the function `foo` using this solution, indeed, reaches the test target.

The static analysis of SE prevents it to deal with dynamic data structures (e.g., unbounded arrays, recursive data structure) and dynamic instructions, such as unbounded loops or recursion. Thus, the efficiency of SE is influenced by the solver limitations, i.e., unsupported data type or instructions that cannot be symbolically evaluated (e.g., native function calls). To mitigate these limitations, Godefroid *et al.* (2005) propose another POA called DSE.

Dynamic Symbolic Execution (DSE)

Due to the limitations of constraint solvers, static SE hasn't been successful in test inputs generation. DSE appeared as a booster of SE at the moment when SE had difficulty to take its place in the field of automatic test-data generation. DSE also called *concolic execution*, was used for the first time in 2003 by Larson et Austin (2003) to strengthen the test-data suite, i.e., generating more than one test-datum for each test target. DSE in its original version consists of running an instrumented version of the UUT with a simple input data (e.g., randomly generated from the input domain) while a symbolic execution extracts the executed path condition. Then, this path condition is solved to generate a new test-data that may be relevant to generate an error (assertion). In 2005 DSE has been improved by Godefroid *et al.* (2005) to generate test data that may discover new execution paths.

```
1  int foo(int x, int y, int z){
2      x=hash(x);
3      int max=x;
4      if(max<y){
5        max=y;
6        if(max<z)
7          max=z;
8      }
9      else
10       if(max<z)
11         max=z;
12     return max;
13 }
```

Figure 2.6: Dynamic symbolic execution: `foo` function is a UUT

They proposed an incremental generation of test data by the hybridization of the symbolic and concrete execution: during the concrete execution a path condition is extracted. The extracted path condition is modified to derive a new one, then the derived path condition is solved to generate a new test-data that may explore a different execution path.

DSE is a dynamic variant of SE. The main difference compared to SE is in the way DSE simplifies complex constraints and explores execution paths: instead of the static exploration of the symbolic tree used in SE, DSE dynamically explores execution paths and generates path conditions.

In the general, DSE typically works as follows:

1. Randomly generate a first test data to start the DSE.

2. Dynamically explore an execution path and generate its path condition using the current test data to execute an instrumented version of the UUT. The condition of the executed path is extracted and complex expressions are replaced with their concrete values;

3. Exit if a stopping condition is met (e.g., the test target is reached).

4. Generate a new path condition: a new path condition is derived from the extracted path condition by negating the last branch condition.

5. Generate a new test data: the new path condition is solved to generate a new test inputs;

6. Replace the test data by the new one and return to step 2.

As example, let us assume that the function `foo` in Figure 2.4 is a UUT and the test requirement is covering all-branches. Let us suppose that the triplet $t_0 = (0, 0, 0)$ is the first

test data that was randomly generated. The execution of the function `foo` using t_0 generates the path condition $cp_0 = \neg(X < Y) \wedge \neg(X < Z)$. To generate a new path condition, the condition cp_0 is derived by negating the last branch condition: $cp_0^{'} = \neg(X < Y) \wedge (X < Z)$. To generate a new test data, the path condition $cp_0^{'}$ is solved by using a CP solver suppose that the solution returned is the triplet $t_1 = (5, 2, 7)$. The execution of the function `foo` using t_1 returns the path condition $cp_1 = \neg(X < Y) \wedge (X < Z)$. Because the last branch condition and its negation are already explored this condition is removed from the path condition and the derived path condition is $cp_1^{'} = (X < Y)$. Let us suppose that the solving of $cp_1^{'}$ returns the triplet $t_2 = (1, 3, 6)$. The execution of the function `foo` using t_2 returns the path condition $cp_2 = (X < Y) \wedge (Y < Z)$. The derived path condition is $cp_2^{'} = (X < Y) \wedge \neg(Y < Z)$. Let us suppose that solving $cp_2^{'}$ returns the triplet $t_3 = (1, 3, 3)$. Thus, the test suite $T = \{t_0, t_1, t_2, t_3\}$ satisfies all-branch requirement for the function `foo`.

With this example we showed how DSE explores execution paths and generates path conditions. To show how DSE simplifies a constraint, let us assume that the function `foo` is modified as in Fig. 2.6: at Line 2, we introduced a new instruction that assigns the value of `hash(x)` to `x`. Because a solver cannot deal with the function call `hash(x)`, the static SE has no chance to generate test data for any test target in this modified example: by replacing `X` with its new value `hash(X)` all path conditions cp_0, cp_1, cp_2, $cp_2{'}$ become complex to solve. To address this problem, instead of the symbolic expression `hash(X)`, DSE uses its concrete value extracted during the execution. Suppose that `hash(0)=10`, the path condition cp_0 become $cp_0 = \neg(10 < Y) \wedge \neg(10 < Z)$.

The main advantage of POA approaches is in the decomposition of the test-data generation problem into sub-problems and solving only one sub-problem at time. On the other hand, the exploration strategy of execution paths is the main disadvantage of the POA. To mitigate some limitations of POA, Gotlieb *et al.* (2000) propose GOA.

2.3.5 Goal Oriented Approach (GOA)

The GOA (Gotlieb *et al.*, 2000; Botella *et al.*, 2002) is an alternative of static SE. It differs from the SE in that it does not subdivide the test-data generation problem into sub-problems, but translates the whole test-data generation problem into one CSP. The resulting CSP is completed with constraints that represent a test target, i.e., a test requirement. A test data is generated by solving the complete CSP. To generate a CSP for a UUT, the GOA works in two stages: (1) the UUT is put in Static Single Assignment Form (SSA-Form) and (2) using guarded constraints, each statement in the SSA-Form is directly translated into a simple or guarded constraint. Both translation steps of the function `foo` are presented in Figures 2.7 and 2.8.

```
1  int foo(int x0, int y0, int z0){
2      int max0−6;
3      max0= x0;
4      if(max0 < y0)
5        max1= y0;
6        if(max1 < z0)
7          max2= z0;
8        max3= φ(max1, max2);
9      }
10     else{
11       if(max0 < z0)
12         max4= z0;
13       max5= φ(max0, max4);
14     }
15      max6= φ(max3, max5);
16     return max6;
17 }
```

Figure 2.7: SSA-form of `foo` function

```
1  Var int foo0, x0, y0, z0;
2  Var int max0−6;
3   max0 = x0;
4
5  (max0 < y0) ⇒ max1 = y0;
6
7  (max0 < y0) ∧ (max1 < z0) ⇒ max2 = z0;
8  (max0 < y0) ∧ (max1 < z0) ⇒ max3 = max2;
9  (max0 < y0) ∧ ¬(max1 < z0) ⇒ max3 = max1;
10
11
12 ¬(max0 < y0) ∧ (max0 < z0) ⇒ max4 = z0;
13 ¬(max0 < y0) ∧ (max0 < z0) ⇒ max5 = max4;
14 ¬(max0 < y0) ∧ ¬(max0 < y0) ⇒ max5 = max0;
15 (max0 < y0) ⇒ max6 = max3;
16 ¬(max0 < y0) ⇒ max6 = max5;
17 foo0 = max6;
```

Figure 2.8: CSP of `foo` function using GOA

A guarded constraint (Jaffar et Maher, 1994) also called conditional constraint, is a constraint composed of two parts: a guard part and a goal, also called the body of the guarded constraint. The addition of the goal to a CSP depends on the state of the former: if the guard is implied (satisfied) by the state of the CSP, then the goal is inserted into the CSP and the guarded constraint is removed; else if the negation of the guard is implied by the state of the CSP then the guarded constraint is removed, otherwise, a solver keeps the guarded constraint in a suspended state.

Static Single Assignment Form (SSA-Form)

The SSA-Form is a formal representation of a program that preserves the program semantics (Cytron *et al.*, 1991; Brandis et Mössenböck, 1994). It defines variables uniquely at each assignment statement and each use can be reached from these definitions. For example, for a variable V, a new derived variable V_i is generated at each new assignment of V encountered in the program. The variable V is replaced by V_i in any statement that uses V until the next definition of V. This renaming process is managed by a σ function: for a statement of a variable X, it returns the number of previous assignment statements of X. In a program that contains branches several derived variables of the same variable can reach the same point. These derived variables must be merged into one variable that is used henceforth. This variable is generated by a φ function: if a control instruction reaches the joint node via branch number j, then the value returned by this function is the value of the operand number j. A φ function is added to the end of each conditional statement. Figure 2.7 shows

```
1   int sePb1(int i, int j){
2       int tab[5]={0,0,0,0,0};
3       tab[j] = 1;
4       if(tab[i]==1)
5       {
6           //test target;
7       }
8   }
```

Figure 2.9: Symbolic execution: aliases problem

```
1   int i : [0,4];
2   int j : [0,4];
3   int tab_0[5]={0,0,0,0,0};
4   int tab_1[5];
5
6   for (int k=0;k<5;k++){
7       k = j ⇒ tab_1[j] = 1;
8       k ≠ j ⇒ tab_1[j] = tab_0[j];
9   }
10  tab_1[i] = 1 ⇒ test target
11
12  tab_1[i] = 1
```

Figure 2.10: GOA model to reach test target

the SSA-Form of the function `foo`

As example, let us suppose that the function `foo` is a UUT and the test requirement is reaching Line 6. This line is translated into Line 7 in the SSA-Form presented in Figure 2.7. Line 7 is control-dependent on the branch conditions at Line 6 and at Line 4. Therefore, the constraint equivalent to the test target is $tc = (max_0 < y_0) \wedge (max_1 < z_0)$, i.e., the conjunction of the branch conditions on which the test target is control-dependent. The resolution of the conjunction tc and the CSP of foo shown in Figure 2.8 leads to a test input that can reach Line 6 in the function `foo`.

The advantages of GOA compared to SE are avoiding the expensive cost of the symbolic evaluation and dealing efficiently with bounded arrays. For example, Figure 2.9 presents the SE aliases problem: to reach the test target at Line 6, `tab[i]` and `tab[j]` must refer to the same storage location, i.e., `i` and `j` must be equal, but SE cannot deduce this reality, consequently it cannot reach the test target at Line 6. Contrary to SE, GOA can implicitly deal with this problem: translating the test requirement of the testing problem presented in Figure 2.9 yields the CSP in Figure. 2.10. Any CP solver will return a solution for this model where `i` equals `j`.

The main advantage of SE compared to GOA is in decomposing the problem of test-data generation: SE treats one execution path at a time; this may negatively influence performance in the presence of infeasible paths because SE may try to prove many execution paths that are infeasible for a same reason, e.g., two inconsistent conditions.

CHAPTER 3

RELATED WORK

This chapter surveys previous work related to this research work in ATDG for software, especially work that uses static approaches to improve SB-STDG.

3.1 Constraint Based Software Test Data Generation

CB-STDG is related to our research work in using CP techniques to solve the problem of test-data generation.

3.1.1 Symbolic Execution

In the last decade two test data generation tools have been implemented based on SE: Symbolic PathFinder (SPF) (Păsăreanu et Rungta, 2010) and PEX (Tillmann et de Halleux, 2008).

SPF is an open-source testing tool for Java programs that has been developed by NASA Ames Research Center (Păsăreanu et Rungta, 2010). It is a part of the project Java PathFinder (JPF) Visser *et al.* (2004) which is a model checker. SPF uses lazy initialization (Khurshid *et al.*, 2003) to deal with unbounded or recursive data structures (e.g., linked list). It does not require prior bound on inputs' sizes, rather it initializes a required inputs data-member when it is first accessed during symbolic execution. Also, to deal with the solvers limitations, SPF gives the possibility to switch between several solvers: choco (Choco, 2012) to deal with integers or real data types; cvc3 (Barrett et Tinelli, 2007) to deal with linear constraints; IAsolver (IAsolver, 2012) to deal with interval arithmetic constraints. Switching between solvers is useful as long as any generated path condition can be solved by using only one specific solver. But this solution cannot deal with complex path conditions that need different solvers or involve black-box or unavailable source code.

PEX (Tillmann et de Halleux, 2008) is a test inputs generator for .Net programs that has been developed by Microsoft Research. Its aim is a high code coverage with a test suite as small as possible. PEX keeps track of all generated test inputs and uses them in a meta-strategy to select the next path to be explored, which allows PEX to reduce the exploration cost of execution paths. To address the solver limitation PEX uses the powerful theorem prover Z3 (Z3, 2012) that can deal with several theories (e.g., arithmetic, bitvectors, and arrays). Z3 helps PEX to solve many problems related to the solver used. Thus, PEX

monitors the UUT executions and analyses communications, i.e., exchanged data, between UUT and its environment and generates an under-estimated model of the environment. This model helps PEX to deal with some types of black-box or unavailable source code. However complex environments cannot be treated with such solution. Thus, Z3 is not ideal and cannot efficiently solve some constraints, e.g., non-linear constraints.

SE suffers from two main problems: the scalability problem because it cannot deal with complex instructions and the exploration of a symbolic execution tree which faces the problem of combinatorial explosion because of the exponential growth of the search space. Following, we present research work dealing with these two problems.

3.1.2 Dynamic Symbolic Execution

To mitigate limitations of SE , DSE (Godefroid *et al.*, 2005; Sen et Agha, 2006) uses concrete values that are extracted from an actual execution to simplify any complex constraint. Researchers have explored different means to make symbolic execution more scalable: dealing with pointers, strategies to explore execution paths, solvers to deal with special types of constraints, and combining DSE with SB-STDG.

Scalability problem

CUTE (Sen *et al.*, 2005) generates complex test-data to explore as many achievable paths as possible. It proposes a DSE approach to address the problem of dynamic data structures by generating two separate models of constraints: one over primitive types and another over pointers. To solve the model over pointers, CUTE uses the theory of equality. It support only two types of constraints: equality and disequality (e.g., $x = y$, $x \neq y$). Also, a linear programming solver is used to solve constraints over integers.

To deal with complex expressions Păsăreanu *et al.* (2011) propose an incremental approach to solve a path condition. The proposed approach consists of separating a path condition into two constraints: `SimplePC` and `ComplexPC`. `SimplePC` is the part of the path condition that contains no uninterpreted function, while `ComplexPC` contains only uninterpreted functions (e.g., native function call). The approach solves SimplePC and, by using its solution, a concrete value is computed for each uninterpreted function to simplify `ComplexPC`. `ComplexPC` is simplified and conjuncted with `SimplePC` to generate an input data that explores a new execution path.

Exploration problem of symbolic execution tree

To address the problem of exploring a symbolic execution tree Xie *et al.* (2009) propose a strategy based on a two fitness functions: path fitness function and branch fitness function. The approach determines a distance for each executed path, i.e., measures how far is a path from the test target (e.g., the branches that are not yet covered). A path fitness is equal to the sum of branch distances (Tracey *et al.*, 1998) of all its branches. The path to be derived is selected among all execution paths according to its fitness value. To select the branch to be negated the approach defines a second fitness function. A branch is evaluated in terms of the gain (i.e., path fitness enhancement) that it brought to the derived paths by its previous negations. This approach has been implemented as a plugin to PEX (Tillmann et de Halleux, 2008), applied on 30 medium size programs, and compared with three other strategies of exploration: random, PEX (default), and iterative deepening. Overall, the approach is generally better, but for 8 programs it is that worse than random. This behavior is caused by the proposed fitness function that does not focus on the test target, but tries to reach the test target through the paths that are already covered. Consequently, the exploration may hang in loops infinitely or until the maximum depth of symbolic execution is reached.

To address the problem of exploring a symbolic execution tree Staats et Păsăreanu (2010) propose a strategy based on parallelism. The key idea behind their approach is partitioning the symbolic execution tree into balanced partitions, then using a worker (task) to explore each partition. The approach has been implemented as a plugin to Java Pathfinder (Visser *et al.*, 2004) and applied on a set of selected programs. The results show that the acceleration of the execution time is correlated with the number of tasks used (workers) and can reach 90 %. The approach requires a complete generation of the symbolic execution tree that may be expensive in the presence of loops and recursion. Also, in some cases only one partition may contain the exploration problem, i.e., a worker may meet the exploration problem and hangs indefinitely.

Many other tools have implemented DSE in its basic version and thus inherited its limitations. PathCrawler (Williams *et al.*, 2005) and Osmose (Bardin et Herrmann, 2008) are two test data generation tools, which can generate a test data to cover all paths (PathCrawler for C programs, Osmose for executables). The limitations mentioned above make these tools useful only with restricted types of programs.

All the approaches mentioned in this subsection or the DSE in general replace a complex expression or a uninterpreted function (i.e., unsupported function call by the solver) by its concrete value of a previous execution, then it simplifies the constraint. This manipulation can generate infeasible constraint, consequently it may miss the exploration of some execution

paths and decrease code coverage.

3.1.3 Goal Oriented Approach

INKA (Gotlieb *et al.*, 2000; Botella *et al.*, 2002) is the first tool that implemented GOA to generate test data for C programs. Yet INKA's approach is independent of the programming language. Although this approach can reach any instruction in a program it cannot be automated to generate test data for a given coverage criterion. Contrary to the POA, GOA has difficulty in identifying the required program instructions to achieve a given coverage criterion.

Euclide (Bardin *et al.*, 2009; Gotlieb, 2009) is a testing tool based on GOA. To address the problem of identifying required instructions to achieve a given coverage criterion, Euclide implements several algorithms to support all-statements or all-decisions coverage criteria, but with an explicit identification of instructions to reach in the code. However this tool without an appropriate algorithm cannot generate a test data to cover other coverage criteria, such as all-paths.

3.1.4 Summary

CB-STDG is appropriate when the feasibility of an execution path can be determined from the source code only. It is not appropriate when a program depends on its environment and involves complex instructions (e.g., uninterpreted function). Besides, scalability is another major problem that CB-STDG approaches faces.

3.2 Search Based Software Test Data Generation

SB-STDG approaches are related to our dissertation because we propose different static analyses to improve them.

Many researchers have considered random search and genetic algorithms as potential search heuristics for test-data generation (Michael *et al.*, 2001; McMinn *et al.*, 2006; Tracey *et al.*, 2000; Harman et McMinn, 2010; Baresel *et al.*, 2002; Wegener *et al.*, 2001; Parasoft, 2013; Andrews *et al.*, 2006; Csallner et Smaragdakis, 2004; Pacheco et Ernst, 2005; Pacheco *et al.*, 2007; Oriat, 2005). We distinguish three broad categories of SB-STDG: random-testing approaches, evolutionary-testing approaches, using static analyses to improve SB-STDG.

3.2.1 Random Testing

Because random testing scales to large systems, random test-data generation is a widely used approach. It was explored in several works to generate test data that meets different coverage

criteria (Parasoft, 2013; Andrews *et al.*, 2006; Csallner et Smaragdakis, 2004; Pacheco et Ernst, 2005; Pacheco *et al.*, 2007; Oriat, 2005). JTest (Parasoft, 2013) is a commercial tool that uses random testing to generate test data that meets structural coverage. Jartege (Oriat, 2005) randomly generates test driver programs to cover Java classes specified in JML. RUET-J (Andrews *et al.*, 2011) is an enhanced version of Jartege that adds some features, such as the minimization of a test case suite. JCrasher (Csallner et Smaragdakis, 2004) generates test data that is susceptible to detect program crashes by performing a random search. Eclat and RANDOOP (Pacheco et Ernst, 2005; Pacheco *et al.*, 2007) use random search to generate test data that are likely to expose fault. To boost the random search, these two tools use a dynamic pruning approach: they prune sequences of methods that violate some predefined contracts. All these works use random search without sufficient guidance, therefore they typically achieve low code coverage.

Zhang *et al.* (2011) use information from a dynamic analysis to sequences and use information from a static analysis to diversify the generated sequences using relationship between methods. Their static analysis focuses on methods and considers that two methods are related if the data members they read or write overlap. Effectively, this static analysis helps to explore new program behaviors and states. However, static analyses may be more relevant for SB-STDG if focused on different features of UUT and used to generate sequences or reduce the search space.

3.2.2 Evolutionary Testing

To improve over random testing, global and local search algorithms have been implemented in several ways. eToc (Tonella, 2004) is a pioneering tool that uses genetic algorithms to generate test data that meet some structural criteria. It only deals with primitive types and strings and, since its creation in 2004, it has not been maintained to better exploit the strengths of recent testing approaches. EvoSuite (Fraser et Arcuri, 2011, 2013c) is also a tool that automates test case generation using GA. Its objective is to achieve a high code coverage with a small test suite. To achieve this objective, EvoSuite integrates recent state-of-the-art approaches.

3.2.3 Static Analyses

Korel (1990) propose a dynamic data-flow analysis approach for path coverage. His approach consists of analyzing the influence of each input variable on the successful sub-path traversed during program execution. During a genetic evolution, only input variables that influence the successful sub-path have their values changed.

Harman *et al.* (2007); McMinn *et al.* (2012a) studied the impact of search-space reduction on search-based test-data generation. In their study, for a given test-data generation problem (e.g., a branch), they use a static-analysis approach (Harman *et al.*, 2002; Binkley et Harman, 2004) to remove irrelevant input variables from the search space. The analyses proposed in these approaches target procedural programming and focus on the arguments of a function or procedure under test.

Ribeiro *et al.* (2008) use a purity analysis to reduce the size of the search space for SB-STDG in OOP: all pure methods are discarded while generating the sequences of method calls, i.e., pure methods are considered irrelevant to the test case generation process. However, many impure methods may also be irrelevant; for example, a class containing a hundred public methods that are impure just because they all invoke an impure public method m. The method m is the most relevant method. Therefore, the purity analysis may reduce the search space but still needs additional analysis to refine the set of relevant methods by keeping only accessible methods that are at the roots of the tree of relevance. Further, the analysis proposed by Ribeiro *et al.* (2008) only generates a set of relevant methods for the CUT.

An evolutionary testing schema theory has been previously defined and analyzed by Harman et McMinn (2010). They define an evolutionary testing schema as a set of individuals that satisfy a constraint (branch) and whose order is the number of genes, i.e., arguments, involved in the constraint. This means that all schemata may have a same order (e.g., a input vector composed of one argument) which violates the building-blocks hypothesis that underlies schema theory. The building-blocks hypothesis supposes that there exist a set of schemata composed of a different number of building-block, i.e., have different order, and the combination of lower-order schemata may build a fitter and higher-order schema. In addition, instead of defining a fitness function that reflects their schemata they propose the use of the approach level and branch distance measure to compute the fitness value of a schema using, whereas branch distance cannot be evaluated for a none executed branch (a schema). Therefore, their schemata concept is useful to analyze only some particular cases of test-data generation problem. In this research work, we adapt schema theory to analyze and improve evolutionary testing for any test-data generation problem.

Using Seeding Strategies

Seeding strategy is a static analysis that focuses on constants appearing in the source code. Many works propose different seeding strategies to improve SB-STDG (Alshraideh et Bottaci, 2006; McMinn *et al.*, 2010; Alshahwan et Harman, 2011; Fraser et Zeller, 2011; Fraser et Arcuri, 2012; McMinn *et al.*, 2012b). When a branch condition involves constants, covering such a branch may be challenging but it may become easier to reach if the set of constants

exist in the source code used when generating instances of classes. Alshraideh et Bottaci (2006) propose a seeding strategy and show that the seeding of string literals extracted from the source code during the generation of instances may reach better level of code coverage than a good fitness function. Alshahwan et Harman (2011) propose a dynamic seeding strategy for Web applications by extracting constants constructed by the application as it executes. Fraser et Arcuri (2012) study different seeding strategies and show that the use of an adequate strategy can significantly improve the SB-STDG. To cover branches involving strings, McMinn *et al.* (2012b) propose a seeding strategy that extracts string literals from the results of Web queries.

The main difference among previous works is in using different sources of constants. All previous approaches use a constant probability of seeding and seed either primitive types or strings. Fraser et Arcuri (2012) show that a constant probability equal to 0.2 gave best results compared to other values but this probability may be harmful in some cases. Indeed, if the extracted set of constants contains only one value, it is undesirable to have 20% of a population formed of the same value because it substantially reduces diversity. Therefore, it is preferable to use a seeding strategy with a variable probability.

3.2.4 Summary

SB-STDG approaches scale well, but they are very sensitive to their guidance. It is inefficient to use a SB-STDG approach in a large search space without sufficient guidance. Generally, when the SB-STDG guidance is discussed, only the fitness function is considered, although the ways that test-data candidates initially generated and modified during the search are another potential means to guide SB-STDG. Also, conditional statements are typically considered as the main source of information for guidance, although a source code is a rich body of information, which can be analyzed and used to amplify the guidance (e.g., the way that data members used in the source code). Previous work mainly focus on fitness function to guide SB-STDG and few works use seeding strategies or sequences generation to amplify the guidance of SB-STDG through the way that test-data candidates generated.

3.3 Combining SB-STDG and CB-STDG

Because, in this research work, we propose CSB-STDG an approach that combines SB-STDG and CB-STDG the following paragraphs present and discuss previous approaches that combine SB-STDG and CB-STDG.

EVACON (Inkumsah et Xie, 2007) was the first tool to combine a CB-STDG approach and a SB-STDG approach. It bridges eToc (Tonella, 2004) and jCute (Sen et Agha, 2006)

to generate test data for unit class testing. eToc is used to generate sequences of method calls and jCute is used to generate values of arguments. In this approach, CB-STDG and SB-STDG work in cooperation: each of them has a specific task to achieve. Thus, if one approach fails in its task (e.g., CB-STDG cannot generate arguments' values because the path condition contains uninterpreted function) the other one cannot help and the whole process of test data generation will fail. Therefore, this approach suffers from limitations of both approaches.

To solve constraints over floating point, the CB-STDG tool PEX (Tillmann et de Halleux, 2008) has been extended by using FloPSy (Lakhotia *et al.*, 2010), which is a SB-STDG approach that solves floating point constraints. FloPSy deals with a specific issue by using SB-STDG to help CB-STDG in solving constraints over floating point instructions. But obtained results compared to a dedicated solver were not promising.

Baars *et al.* (2011) propose a fitness function based on symbolic execution. The proposed fitness function analyses and approximates symbolic execution paths: some variables are approximated and any branches involving these variables are ignored. Then, the fitness value is computed based on the number of ignored conditions and the path distance (branch distance). This fitness function uses constraint programming to enhance SB-STDG.

Malburg et Fraser (2011) propose a hybrid approach that combines GA and DSE on Java PathFinder. This approach uses GA to generate the test inputs; during the GA evolution the approach calls CB-STDG to explore a new branch of the program. CB-STDG is used as a mutation operator that uses SE to derive a constraint path. Then it uses DSE to solve this constraint. Therefore, the test data that is generated based on the combination is actually generated using DSE. As explained in Section 2.3.4, DSE uses concrete values to simplify complex expressions. It is well-known that these concrete values may make the path constraint infeasible. In general, these concrete values are randomly chosen, in which case DSE falls in a random search.

All approaches proposed to use CB-STDG or combine it with SB-STDG apply CB-STDG on a whole UUT (Inkumsah et Xie, 2007; Lakhotia *et al.*, 2010; Malburg et Fraser, 2011). Therefore, these approaches are limited by the size and the complexity of the UUT and the fact that the data-test generation problem is undecidable.

Part II

CP Techniques to analyze and improve SB-STDG

CHAPTER 4

CP APPROACH FOR SOFTWARE TEST DATA GENERATION

The main aim of structural testing is to find the set of test data that can satisfy a given coverage criterion, i.e., reaching some specific statements in the source code (e.g., statements, branches, or paths). To achieve this aim, either SB-STDG or CB-STDG explore execution paths according to a strategy: dynamically by executing an instrumented version of the source code or statically by systematically or heuristically exploring the symbolic execution tree. In this chapter, we present CPA-STDG, an approach that uses the techniques implemented in CP solvers to explore execution paths in an efficient way. The approach models the CFG and the implementation of a UUT in two different CSP then links them into one main CSP. The integration of CFG modeling has several advantages over previous work, i.e., SE and GOA: (1) it facilitates identifying the program statements required to satisfy a coverage criterion; (2) it reduces the number of infeasible execution paths to explore; and, (3) it can answer to different coverage criterion. CPA-STDG starts by modeling a program under test and its CFG as a main constraint satisfaction problem. Then, a test coverage criterion is formulated as a set of constraints (we call it, criterion-constraints) or a search heuristic. Each constraint in the criterion-constraints is solved in conjunction with the main CSP, yielding a test-data suite or proving the infeasibility of this constraint, i.e., a part of the UUT is unreachable. We assume as input a function or procedure under test and a structural criterion to cover. The approach produces a test-data suite, a set of covered criterion-constraints, and a set of uncovered criterion-constraints if any.

We empirically evaluate CPA-STDG on different benchmarks to compare code coverage and time required to generate test data with the state-of-the-art CB-STDG techniques. As state-of-the-art CB-STDG techniques, we choose PathCrawler (Williams *et al.*, 2005; Botella *et al.*, 2009), a representative of the POA family of techniques, and INKA (Gotlieb *et al.*, 2000) a representative of the GOA family of techniques.

4.1 Modeling Constraints of a Program and its CFG

CFG nodes and control dependency between them are the basis of our methodology to model a program with its CFG by a CSP: each CFG node and variable declaration generates a CSP variable; some control dependency relationships are translated into equivalence constraints; and, each statement or conditional expression is translated into a guarded constraint.

To translate a program using our approach, we propose three main steps: the SSA-Form model; CFG constraints modeling; and the construction of the global CSP.

If a program under test contains two assignment statements of the same variable to two different values (e.g., $x = 1$ and $x = 2$), it generates two inconsistent constraints. To avoid this well known problem Gotlieb *et al.* (2000); Gotlieb (2009); Collavizza et Rueher (2006), before translating the program to a CSP, we put it in a SSA-Form model.

4.1.1 Modeling Constraints of CFG

This step consists of modeling a program's CFG as a preliminary CSP. We begin by generating the CFG. We distinguish between two classes of CFG nodes: decision-nodes and statement-nodes. *A decision-node* is a node that has two outgoing arrows (i.e. it represents a conditional statement). *A statement-node* is a node that accepts one incoming arrow and one outgoing arrow (i.e., it represents a basic block that does not contain any conditional statement). After generating the CFG, each node is codified according to its class and its order in the the program: a decision-node is identified by a prefix D and an index i and a statement-node is identified by a prefix S and an index j. After identifying nodes, we label edges according to the original node: an edge outgoing from a statement-node is labeling by 1, an edge outgoing from a decision-node is labeled by 1 if it represents the true branch and it is labeled by -1 it represents the false branch. During the generation of a CSP, a node is translated into a variable whose domain is the set of labels of its outgoing edges and the value 0 that represents that the node does not belong to the current execution path.

Decision-node

A decision-node is translated into an CSP integer variable with a small domain {-1, 0, 1}: -1 means that the false branch of a node is a part of the current execution path; 0 means that a node is not a part of the current execution path; 1 means that the true branch of a node is a part of the current execution path.

Statement-node

A statement-node is translated into an CSP integer variable with a small domain {0, 1}: 0 means that a node is not a part of the current execution path; 1 means that a node is a part of the current execution path. A statement-node that makes a part of all execution paths must be equal 1.

Control dependency relationship

An immediate dominator Ferrante *et al.* (1987) is a relationship between two CFG nodes D_i and N_j. D_i *dominates* N_j iff D_i is a decision-node that appears in all paths from START to N_j. D_i is an immediate dominator of N_j iff it is the closest strict dominator on any path from START to N_j.

If D_i is an immediate dominator of N_j, then this relationship is translated into two constraints:

$$D_i \neq b \Leftrightarrow N_j = 0, \text{where b equal to 1 or -1 depending on the branch of } D_i \text{ that dominates } N_j. \tag{4.1}$$

$$D_i = b \Leftrightarrow N_j \neq 0, \text{where b equal to 1 or -1 depending on the branch of } D_i \text{ that dominates } N_j. \tag{4.2}$$

These constraints model the logic of an execution path. The constraint 4.1 means that if the immediate dominator does not belong to the current execution path, then the node cannot be a part of the current execution path. The constraint 4.2 means that the node can be a part of the current execution path iif its dominator belong to the current execution path.

Note, that if we fix a given node (e.g., test target that aims to reach a given branch or a statement), the propagation will fix all nodes within this node is control dependent. Thus, one path is explored at time, only real execution path are explored, and if a part of a path is proved infeasible none path contains this part will be explored.

Conditional Statement with Multi-clauses

The Modified Condition/decision coverage criterion expresses its requirement in terms of clauses in stead of CFG components. This coverage criterion requires more details that make the decomposition of multi-clauses condition mandatory for identifying its testing requirements.

In order to show hidden branches in a multi-clauses decision-node, from every decision-node, we derive a decision graph by breaking the multi-clauses expression into atomic conditions. A multi-clauses condition uses two forms of logical expressions: conjunctive form or disjunctive form. Each form can be expressed in terms of its atomic conditions. A conjunctive (resp. disjunctive) form is true (resp. false) iff all its atomic conditions are true (resp. false). The relationship between a conjunctive decision node D_i and its derived conditions nodes D_{ij} is formally expressed by the constraints in Table 4.1. The relationship between a disjunctive decision node D_i and its derived conditions nodes D_{ij} is expressed similarly as in conjunctive form. To get the constraints table of disjunctive decomposition, we swap 1 and -1 in Table 4.1.

Table 4.1: Constraints table of a conjunctive condition

D_i	D_{i1}	...	D_{ij}	...	D_{in}
1	1	1	1	1	1
0	0	0	0	0	0
-1	-1	*	*	*	*
⋮	⋮	⋮	⋮	⋮	⋮
-1	*	*	-1	*	*
⋮	⋮	⋮	⋮	⋮	⋮
-1	*	*	*	*	-1

For an atomic decision, it is enough to add only one constraint expressed by the equality between decision node variable D_i and the derived condition node variable D_{i1} ($D_i = D_{i1}$). For complicated expressions that combine conjunctive and disjunctive forms, it is preferable to use intermediate nodes. Note, that the decomposition of decision nodes is not necessary if the testing requirement does not involve the clause level.

To express the relationship between conditional statement and clauses the following constraint must to be inserted CFG constraints model:

$$table(\tau), \text{where } \tau \text{ is the constraints table.} \tag{4.3}$$

The global constraint *table* considers that the table τ contains explicitly the list of allowed tuples (Régin, 2011). Therefore, variables involved in such a constraint must satisfy a tuple in the table τ.

4.1.2 Modeling Constraints of Statements

The third step uses the preliminary CSP, the SSA-Form model, and the relationship between a node and its statements to create a new global CSP. This means translating each statement in the SSA-Form model into one or more constraints. In this section we also discuss some particularities of variable declarations, expression assignments, and loops.

Variable Declaration

A variable declaration of a given variable V_0 of a primitive type T (e.g. integer, character, boolean) is translated into the global CSP by declaring a new variable V_0 of type T. The domain of V_0 is extracted by analyzing the program, it can be bounded between two values V_{min} and V_{max}, otherwise the domain contains all possible values of T (e.g., integer type is defined in the range $[-2^{-31}, 2^{31} - 1]$).

A declaration of a fixed-size array is translated by an array of similar size in the global CSP. A declaration of an array t_0 of variable size (dynamic allocation) is reflected in the global CSP by an array with a bounded size: the maximum size is a predefined constant L_{max} and a variable l_{t0} is used to represents the actual size on an array. To ignore an index exceeding the actual size l_{t0}, we add some constraints to the CSP: these variables are fixed to a constant and are ignored in all constraints.

Conditional Statement

A conditional statement controls the execution flow of a program according to the evaluation of its expression. In the CFG, it is represented by a decision-node. For a given decision-node D_i that represents a conditional statement over a logical expression $lexp$, if its branch true is a part of the current execution path, then its logical $lexp$ is satisfied. This can be expressed by by a guarded constraint as following:

$$D_i = 1 \Rightarrow lexp_{\sigma(lexp)}, \text{where } lexp_{\sigma(lexp)} \text{ represents } lexp \text{ in the SSA-Form.} \tag{4.4}$$

Thus, if the branch false of D_i is a part of the current execution path, then the negation of $lexp$ is satisfied. This can be expressed by by a guarded constraint as following:

$$D_i = -1 \Rightarrow \neg(lexp_{\sigma(lexp)}), \text{where } lexp_{\sigma(lexp)} \text{ represents } lexp \text{ in the SSA-Form.} \tag{4.5}$$

Assignment Statement

An assignment statement assigns an expression to a program variable. Every assignment statement is translated by two guarded constraints: one to represent that an assignment statement is a part of the current execution path and another to evaluate the assigned variable if the statement is not a part of the current execution path.

Each statement-node S_i represents a certain number of assignment statements in a program under test. This relationship can be expressed on S_i and every assignment statement s in its content. If S_i is a part of the current execution path, then s is executed and a equivalent constraint must be satisfied. This can be expressed as a guarded constraint as following:

$$S_i = 1 \Rightarrow s_{\sigma(s)}, \text{where } s_{\sigma(s)} \text{ represents } s \text{ in the SSA-Form.} \tag{4.6}$$

Thus, if S_i is not a part of the current execution path, then the variable v assigned by s must be evaluated. In the latter case the variable v keeps its already assigned value. This can be

expressed by a guarded constraint as following:

$$S_i = 0 \Rightarrow v_{\sigma(v)} = v_{\sigma(v)-1}, \text{where } v_{\sigma(v)} \text{ represents } v \text{ in the SSA-Form at the statement } s. \tag{4.7}$$

Assignment Statement that assigns an Array Item In a program, an array is handled by two groups of statements: reading and writing. Reading an array item is translated as a scalar variable. In SSA-Form, writing an array item generates a new array variable that contains the same items as its predecessor except that new item which contains a new value. Therefore, an assignment of an array item is a lot of simple assignment statements wherein all items are assigned new values and the constraints 4.6 and 4.7 must be inserted for each item in the array. For example, the assignment $t[j] = exp$ is translated into some constraints as following:

$$S_i = 1 \Rightarrow t_{\sigma(t)}[k] = \begin{cases} exp_{\sigma(exp)} & \text{if } k = j \\ \\ t_{\sigma(t)-1}[k] & \text{otherwise} \end{cases} \tag{4.8}$$

$$S_i = 0 \Rightarrow t_{\sigma(t)} = t_{\sigma(t)-1} \tag{4.9}$$

Or more formally as following:

$$\forall 0 \leq k < L_{max}, S_i = 1 \Rightarrow (k = j \Rightarrow t_{\sigma(t)}[k] = exp_{\sigma(exp)}) \wedge (k \neq j \Rightarrow t_{\sigma(t)}[k] = t_{\sigma(t)-1}[k)$$

$$\forall 0 \leq k < L_{max}, S_i = 0 \Rightarrow t_{\sigma(t)}[k] = t_{\sigma(t)-1}[k]$$

Loop

Because CP is a deterministic approach we cannot model a loop with an infinite number of iterations. In general, to deal with this limitation loops are bounded by a maximum number of iterations, i.e., a small constant *k-path* (equal 1, 2, or 3) is used to limit the number of loop iterations. We can model a loop in two different ways: first, we force a loop to stop at most after k-path iterations, in this case some feasible paths may become infeasible; second, we don't force a loop to stop and we model just *k-path* iterations, after which the value of a variable assigned inside a loop is unknown. We use the first way because it is widely used to define coverage criterion.

To model loops as a first step we transform any program loop into a while loop. We use a constant *k-path* to limit the number of iterations in a loop. Then, we unfold a loop into embedded conditional-statements. With this transformations, a loop becomes a set of

conditional-statements and the constraints 4.1, 4.2, 4.4, 4.5, 4.6, and 4.7 can be used to model it. Therefore, loop contains $k+1$ decision-nodes. To force the program to exit the loop at most after k iterations, the last decision node D_k must be always different from the value 1.

For example, The result of translating a loop ($while\ \ (c)\ \ v = exp;$) is as following:

1. $\forall\ 0 < i \leq k$
   - Applying the constraint 4.1 on decision-nodes: $D_{i-1} \neq 1 \Leftrightarrow D_i = 0$;
   - Applying the constraint 4.2 on decision-nodes: $D_{i-1} = 1 \Leftrightarrow D_i \neq 0$;
2. $\forall\ 0 \leq i \leq k$
   - Applying the constraint 4.1 on a decision-node and a statement-node: $D_i \neq 1 \Leftrightarrow S_i = 0$, where S_i is the statement-node that represent s in the CFG at the i^{th} iteration;
   - Applying the constraint 4.2 on a decision-node and a statement-node: $D_i = 1 \Leftrightarrow S_i \neq 0$;
   - Applying the constraint 4.4: $D_i = 1 \Rightarrow c_{\sigma_i(c)}$, where $c_{\sigma_i(c)}$ is the SSA-Form of c at the i^{th} iteration;
   - Applying the constraint 4.5: $D_i = -1 \Rightarrow \neg(c_{\sigma i(c)})$;
   - Applying the constraint 4.6: $S_i = 1 \Rightarrow v_{\sigma_i(v)} = exp_{\sigma_i(exp)}$;
   - Applying the constraint 4.7: $S_i = 0 \Rightarrow v_{\sigma_i(v)} = v_{\sigma_i(v)-1}$;
3. Force the loop to stop at most after k iterations: $D_k \neq 1$.

4.2 Modeling Constraints of Coverage Criteria

The approach generates a main CSP mainly based on guarded constraints, where a guard is expressed only in terms of a CFG node (i.e., integer variable with a narrow domain) and a goal is expressed on another CFG node or a SSA-Form statement (i.e., complex expression). In general, solvers treat a guarded constraint in a special way. A guarded constraint is seen as two separated constraints, a guard constraint and a goal constraint, where the goal constraint is ignored until the solver proves that the guard constraint is satisfied. Therefore, the generated main CSP implicitly forces a solver to solve the problem in two stages: (1) solving the CFG model to select one specific execution path, and (2) solving the statements model to generate test data. Thus, such a CSP has three advantages: first, a solver treats only simple constraints before selecting an execution path and can quickly prove whether a

guard is satisfied or not because the domain of CFG nodes is small; second, a solver cannot explore more than one execution path at a time, which is an implicit decomposition of the problem; third, the generated CSP keeps all the program's structural semantics that we could benefit from to generate test data to cover any structural coverage criterion: A restriction of the main CSP by some additional constraints or by a search strategy can limit and guide the solution to a specific test-data set.

Generally, the aim of a structural test is to run a set of paths that meets a given coverage criterion. This aim can be decomposed into a set of test-targets which are expressed in terms of decision-nodes or statement-nodes. So, a test target can be expressed by a conjunction of criterion-constraints on CSP variables that represent CFG' nodes. Thus, we can represent a test target by a set of pairs $\langle variable, value\rangle$ ($variable = value$), where $variable$ is a CFG' node and $value$ is in {-1,0,1} for a decision-node and in {0,1} for a statement-node. A test target may contains pairs from the same class or combined classes of nodes. A coverage criterion can be expressed by a set of criterion-constraints (e.g., all-statements is expressed by $\{\{\langle S_i, 1\rangle\} \mid 0 \leq i < n$, where n is the number of statement-nodes$\}$). Then, we ask a solver to generate a solution for every criterion-constraint in conjunction with main CSP. If the CSP become unreachable, then we mark this test-target as infeasible. If the solver generate a test-datum, we check if there exist another test-targets covered by this test-data, in which case we remove them from the set of test-targets.

4.2.1 Control-flow Based Criteria

To generate a test-data set that covers a given coverage criterion, we must first generate the set of criterion-constraints. Below, we give some representative sets of criterion-constraints describing some control-flow based criteria.

1. All-statements: each criterion-constraint is a set that contains a single pair composed of a statement node variable and the value 1 e.g., $\{\{\langle S_i, 1\rangle\} \mid 0 \leq i < n$, where n is the number of statement-nodes $\}$.

2. All-decisions: each partial objective is a set of a single pair which is composed of a decision node variable and the value 1 or -1. The goal set contains all possible partial objectives (e.g., $\{\{\langle D_i, 1\rangle\}, \{\langle D_i, -1\rangle\} \mid 0 \leq i < n$, where n is the number of decision-nodes$\}$).

3. In a similar fashion, we can generate test data for all-conditions, all-multiple-conditions and all-conditions/decisions.

4. Modified Condition/decision: We use the algorithm proposed in Foster (1984) to generate the test-targets.

Heuristic for Achieving All-paths Coverage To cover all-paths we can generate a set of test-targets where each test-target is a path. This way seems like exploring all execution paths using a symbolic-execution tree. This means an unreachable CSP to prove for each infeasible path that can be expensive. Instead of this way of exploration, we propose to use the main CSP with a search heuristic to generate test-data that cover all-paths. In our approach, an execution path is an assignment of decision nodes. A first path can be covered by the first solution reached by solving the main CSP. From this solution, we can construct the path condition of this first path as the conjunction of equality logical expressions between decision-nodes and their values ($D_i = v_i$). The insertion of the negation of this path condition in the main CSP forces the solver to give another solution corresponding to a different path. We continue using this mechanism: for every solution found, the negation of its path condition is inserted in the main CSP, until the CSP becomes unreachable, which means that all feasible paths are covered.

The described search heuristic can be implemented easily in a solver as follow: we define for the main CSP a search strategy that begins by enumerating the decision-nodes (D_i) on the list of input parameters. Once a solution is found by the solver, the strategy backtracks to the nearest node-decisions in the search tree.

Proof of Program Properties The proposed approach can be used to prove some program properties. For example, to prove a post-condition, we formulate the equivalent constraint for this condition by replacing program variables with their equivalent variables in the main CSP, then we solve the conjunction of the main CSP and the negation of this constraint, if the CSP reachable, then there exists a path where the program violates the post-condition. The solver returns the concerned execution path (i.e., an assignment of decision-nodes) and the test datum that breaks the post-condition. If the CSP is unreachable, then there is no path in the program that violates the post-condition.

4.2.2 Data-flow Based Criteria

The modeling method proposed in the previous sections allows considering all paths going through any test target in the same CSP model. Solving such a generated model with constraints that describe a data-flow test-target may generate a test-data to reach it or prove that a data-flow test-target is unreachable which was impossible without the use of constraint programming. Thus, the main aim of this subsection is analyzing a data-flow test-target and

modeling it using constraints.

Informally, the key of any data-flow coverage criterion is the notion of def-use association, which is the relation between two CFG nodes: the first one defines a program variable; while the second uses that variable. A data-flow criterion is represented by a subset of all def-use associations. Test-data generation for a data-flow criterion consists of identifying an adequate set of def-use associations and their corresponding test-data set. Thus, test-data generation for data-flow based criteria can be separated into two sub-problems: first, data-flow analysis, which is concerned with the identification of def-use associations; second, test-data generation, which is concerned with the selection of a set of def-use associations that corresponds to the desired adequacy criterion and the generation of a test-data set that satisfies this def-use set.

A def-use association can be formalized as a criterion-constraint over the main CSP variables. Thus a criterion is a set of criterion-constraints that characterize subpaths needed to be executed.

Before analyzing the data-flow and giving a formal constraints that represents def-use association, we introduce some relevant definitions and notations:

- In the rest of this chapter, we note a def-use association $\langle v, n_d, n_u \rangle$ by t.
- We denote by $D(n)$ (resp. $U(n)$) the set of variables definitions (resp. uses) at node n.
- We denote by $D(v)$ (resp. $U(v)$) the set of definitions (resp. uses) of the variable v.
- A *predecessor* of a node n_j is a node n_i that precedes n_j in an execution path (in a top-bottom traversal of the CFG) Allen (1970). n_j is a *successor* of n_i.
- A predecessor (resp. successor) is an $immediate - predecessor$ (resp. $immediate - successor$) Allen (1970) iff it precedes (resp. follows) immediately that node.
- We denote the set of predecessors by $P(n_i)$, the set of immediate-predecessors by $P^*(n_i)$, and the set of immediate-successors by $S^*(n_i)$.

$$P(n_j) = \bigcup_{n_i \in P^*(n_j)} P(n_i) \cup \{n_i\}$$

The first step of data-flow based testing is the process of def-use analysis that determines the set of all def-uses. This set can be determined using defs, uses, and an adapted version of the algorithm Basic Reach Algorithm Allen et Cocke (1976). In this work, we assume that the CFG does not contain cycles (loops are unfolded). Parsing the CFG from the node START to the node END is enough to determine defs that can reach every node. The algorithm takes

as input four parameters: D is the set of all definitions for every node; U is the set of all uses for every node; P^* is the set of all immediate-predecessor for every node; S^* is the set of all immediate-successors for every node. The algorithm returns three sets: the first contains the set of predecessors for each node, P; the second contains the set of definitions that can reach each node, it is denoted by R; the third contains all def-uses associations denoted by DU. In order to generate these three sets the algorithm uses a set of auxiliary structures: $A(n_i)$ is the set of available definitions immediately after a node n_i; $Ps(n_i)$ is the set of preserved definition after the node n_i, which is the intersection of $R(n_i)$ and $A(n_i)$. For a decision-node or a join-node, these three auxiliary sets are equal. The algorithm for generating the DU, R and P from a control flow graph information is given in Algo 1. The expressions of the formula to generate each set is given below:

- $R(n_j) = \bigcup_{n_i \in P^*(n_j)} A(n_i)$
- $Ps(n_j) = R(n_j) - \{\langle v, n_i\rangle | \langle v, n_j\rangle \in D(n_j)\}$
- $A(n_j) = Ps(n_j) \cup D(n_j)$

Using the R and U sets, the set of def-use associations is constructed according to the following formula:

$DU(n_u) = \{t | \langle v, n_u\rangle \in U(n_u) \wedge \langle v, n_d\rangle \in R(n_u)\}$

The distinction between c-use and p-use is done according to the class of n_u: if n_u is a decision-node so $DU(n_u)$ is a set of p-use; else it is a set of c-use.

A def-use association is represented by a triplet t where the variable v is defined at n_d, used at n_u, and there are paths from n_d to n_u that do not contains any killer-node. To exercise that def-use association, the test data must trigger an execution path that passes through n_d and n_u and not via any killer-node. We denote by $K(t)$ the set of killer-nodes, i.e., the set of statement-nodes that are located between the nodes n_d and n_u and define the variable v. Formally, this set can be represented as follow.

$$K(t) = \{n_i | (n_i \neq n_d) \wedge (n_i \in P(n_u) - P(n_d)) \wedge (\langle v, n_i\rangle \in D(v))\}$$

Using these definitions, a def-use association can be expressed as one or two logical expressions. The following expression is a generator of criterion-constraints. It generates one criterion-constraint if the use node is a statement-node and two criterion-constraints if the use node is a decision-node.

$$criterionConstraints(t) = \{ \bigwedge_{n_i \in K(t)} (n_i = 0) \wedge (n_d = 1) \wedge (n_u = val)\}$$

Algorithm 1 A data flow analysis algorithm.

Input: D, U, S^*, P^*
Output: P, R, DU

```
1: for (each n in CFG) do
2:     P(n) ← nil
3:     R(n) ← nil
4:     NbrP*(n) ← 0
5: end for
6: stack S
7: S.push(START)
8: while (!S.Empty()) do
9:     n ← S.pop()
10:    Ps(n) ← R(n)
11:    for (each ⟨v, Nd⟩ in R(n)) do
12:        if (⟨v, Nd⟩ in D(n)) then
13:            Ps(n) ← Ps(n) − {⟨v, Nd⟩}
14:        end if
15:    end for
16:    A(n) ← Ps(n) ∪ D(n)
17:    for (each Ns in S*(n)) do
18:        P(Ns) ← P(Ns) ∪ P(n) ∪ {n}
19:        R(Ns) ← R(Ns) ∪ A(n)
20:        NbrP*(Ns) + +
21:        if (P*(Ns).size() == NbrP*(Ns)) then
22:            S.push(Ns)
23:            DU(Ns) ← Intersection(R(Ns), U(Ns))
24:        end if
25:    end for
26: end while
```

If n_u is a statement-node, one criterion-constraint is generated in which the clause $(n_u = val)$ is evaluated to $(n_u = 1)$. If n_u is a decision-node, a second criterion-constraint is generated in which the clause $(n_u = val)$ is evaluated to $(n_u = -1)$. This means that the use is a p-use, so that both its edges must be exercised.

With the above expression, it is possible to formalize every def-use association as one or two criterion-constraints, so a data-flow adequacy criterion can be represented by a set of criterion-constraints that characterizes def-uses and needed to be exercised. To obtain a set of test input that satisfies a data-flow adequacy criterion, it is enough to solve each criterion-constraint in conjunction with the Main CSP $(solve(criterionConstraints(t) \wedge MainCSP))$, which generates a test data or proves that some infeasible use-defs or def-clear paths are infeasible. Below we give a representative set of criterion-constraints for the most-used data-

flow adequacy criteria Rapps et Weyuker (1985).

- All-defs criterion is satisfied if the set of test data exercises at least one def-clear path from every definition to its uses.

$$\{\bigvee_{t \in DU} criterionConstraints(t) | \langle v, n_d \rangle \in D\}$$

- All-uses criterion is satisfied if the set of test data exercises at least one def-clear path for every def-use associations.

$$\{criterionConstraints(t) | t \in DU\}$$

- All-c-uses criterion is satisfied if the set of test data exercises at least one def-clear path for every def-use associations that have a statement-node as use node.

$$\{criterionConstraints(t) | (n_u \text{ is a statement node}) \wedge t \in DU\}$$

- All-c-uses/some-p-uses criterion is satisfied if the set of test data satisfies All-c-uses or in case a definition does not have any c-use, the test data exercises a def-clear path from this definition to some of its p-use.

$$\{criterionConstraints(t) | (n_u \text{ is a statement node}) \wedge t \in DU\}$$

$$\bigcup \{\bigvee_{t \in DU} criterionConstraints(t) | c - uses(\langle v, n_d \rangle) = \emptyset\}$$

- All-p-uses criterion is satisfied if the set of test data exercises at least one def-clear path for every def-use association that has a condition-node as use node.

$$\{criterionConstraints(t) | (n_u \text{ is a decision node}) \wedge t \in DU\}$$

- All-p-uses/some-c-uses criterion is satisfied if the set of test data satisfies All-c-uses, in case a definition does not have any p-use, the test data exercises a def-clear path from this definition to some of its c-use.

$$\{criterionConstraints(t) | (n_u \text{ is a decision node}) \wedge t \in DU\}$$

$$\bigcup \{\bigvee_{t \in DU} criterionConstraints(t) | p - uses(\langle v, n_d \rangle) = \emptyset\}$$

- All-du-paths criterion is satisfied if the set of test data exercises all def-clear paths.

$$\{criterionConstraints(t) \wedge def\_clear_i | t \in DU \wedge def\_clear_i \in def\_clear(t)\}$$

where $def\_clear(t)$ is the set of all def-clear paths between the nodes N_d and N_u according to the variable v. It is possible to get this set by tracking def-clear paths Algo 1 but, in this work, for the All-du-paths criterion, we use a search strategy that generates a data test for each def-clear path, which we detail in the next subsection.

Solving Constraints to Generate Test Data

Using the data-flow information, the search phase is split into three sub-phases called levels: both the first and the second levels orient the search using a subset of the decision-nodes variables because they have a small search space (variable domains are small: $\{-1, 0, 1\}$). The third level orients the search using the program parameters and it has a large search space (in general parameter domains are large). The first and the second levels allow to identify the current execution path and to introduce, in a constraint store, goals of conditional constraints that are conditioned by an assignment of decision-node variables. The third level looks for an assignment of program parameters that can satisfy the chosen path. The search process at the third level finishes in either two ways: first, a solutions is reached and stored with its path (nodes assignment); second, a path is partially or completely proved as infeasible and is stored as an assignment of a subset of nodes.

At the first level, the search process behavior depends on the selected coverage criterion. It labels decision-node variables that are located between the def-node n_d and the use-node n_u: variables that represent the decision-nodes that belong to the intersection of both predecessor sets. Depending on the selected criterion, the first level defines its strategy that generates either one test input for each criterion-constraint or one test input for each path in its search tree that covers all def-clear paths. Using this latter strategy, All-du-paths is satisfied with the same set if criterion-constraints as All-uses except that All-uses uses the first strategy.

As seen in Subsection 4.2.1, a data-flow adequacy criterion is a set of criterion-constraints where each criterion-constraint is an assignment of some main CSP variables (nodes). In general, more than one criterion-constraint can be satisfied with the same test data. To avoid test data redundancy, before trying to solve any criterion-constraint, we check if there exists a node assignment, in the generated-solution (resp. proved infeasible path) set, that satisfies this criterion-constraint (resp. proves its unreachablity). If so we stop the solving process before the search phase, which is the most expensive part of the CSP solving process. This checking procedure includes: a static verification before starting the search and a dynamic

verification inside the search process at the end of the first level or at the end of the second level.

4.3 Empirical Study

This section presents our evaluation of the proposed approach that explores execution-paths using a CP solver for an efficient test-data generation. We investigate the advantages and limitations of CPA-STDG by applying it on a set of different benchmarks and comparing it to two different approaches (Gotlieb *et al.*, 2000; Williams *et al.*, 2005; Botella *et al.*, 2009) from the literature that also use a CP to generate test data. In a first case study we apply our approach on a set of different benchmarks to show its applicability, usefulness, and efficiency of our approach for data-flow based criteria. In a second case study we compare CPA-STDG to two different approaches interesting in control-flow based criteria: using the criterion all-branch coverage (Gotlieb *et al.*, 2000) and using the criterion all-path coverage (Williams *et al.*, 2005; Botella *et al.*, 2009).

4.3.1 Case Study 1: Data-flow Based Criteria

We follow the Goal Question Metric (Basili et Weiss, 1984) approach to show the applicability, usefulness, and efficiency of our approach.

The context of our research includes six UUTs: four benchmarks from the testing literature and one well-known algorithm the *Dijkstra* (Ford, 2007). Table 4.2 summarizes the six UUTs used in this study. Each line presents one of the UUT while columns show the maximum number of iterations in a loop (kPath), the number of lines of code (LOC), the number of linearly independent paths (LIP) the number of feasible execution paths that are detected experimentally using a small domain, and the number of def-use associations.

Dijkstra consists of finding the shortest path among a set of paths. It takes as input a graph represented as a bi-dimensional array, the array dimension, and a vertex source. It

Table 4.2: Experimental subjects.

UUT	kPath	#LOC	#LIP	#Feasible Paths	#Def-use
Dijkstra	2	53	2501	29	232
Triangle		36	32	14	40
Tritype		34	57	10	52
Merge	2	29	72	17	64
Sample	2	18	24	20	52
Bsearch	2	18	10	8	46

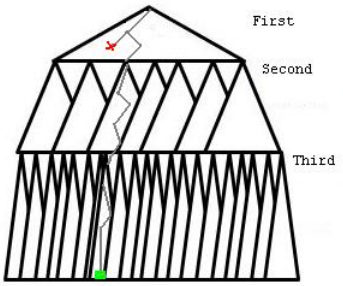

Figure 4.1: Search-heuristics scheme

returns an array that contains the shortest path from the source to any other vertex in the graph. Its implementation contains a large number of infeasible paths and it uses overlapped loops, which increases its cyclomatic complexity and its number of linearly-independent paths (McCabe, 1976).

The others UUTs are well-known benchmarks in the CB-STDG literature: *Tritype*, *Merge*, *Bsearch*, and *Sample* (Collavizza et Rueher, 2006; Gotlieb *et al.*, 2000; McMinn, 2004; Williams *et al.*, 2005). In the literature, Tritype is available in two different implementations: we denote them by Triangle (McMinn, 2004) and Tritype (Collavizza et Rueher, 2006). Merge is the merge part of the merge sort algorithm. It takes as input four parameters, two sorted arrays and their lengths, and returns a sorted array. Sample is a program that compares a reference value to contents of two arrays using two sequential loops; each of them iterates a fixed number of iterations corresponding to the array length. Bsearch is a function that takes as input an array and a value. It return true if the value exist in the array otherwise it returns false. It uses a dichotomy search implemented in one loop.

Research Questions

This cas study aims at answering the following two research questions:

RQ1: Does our approach can generate test data for any data-flow coverage criterion in a efficient way? This question shows the applicability and the usefulness of our approach.

RQ2: Does the proposed search strategy and checking process enhance or weaken the performance of the proposed approach? This question shows the efficiency of our approach.

Analysis Methods

To answer RQ1 and RQ2, we have implemented our approach and applied it on the six UUTs.

For each UUT and data-flow coverage criterion, we generate test data set and we observe some indicators: the number of def-use associations needed to cover; the number of test data generated; the number of def-use associations covered; the number of infeasible clear-path proved; the number of infeasible clear-path avoided (i.e. an infeasible clear-path was used to prune the search process); and the execution time.

We consider the execution time and the ratio between the number of test data generated and def-use associations covered as indicators of efficiency. Thus, the number of infeasible clear-path reused is an indicator of efficiency.

Parameters

To analyze the performance of our approach, we fix the domain of integer to $[-10^6, 10^6]$ for all the input variables. The maximum number of iterations in a loop and the maximum length of an array are fixed to 2 (k-path = 2). During the search a test target is considered infeasible either if the solver completes the search without finding a solution or for the UUT Tritype the number of fails reaches 1000.

All experiments were performed using ILOG Solver 6.7 (IBM, 2009b), Microsoft Visual C++ 2008, on Windows XP, Intel Core 2 Duo 2GHz, 2Go of memory.

Empirical Study Results

We now present the results of our empirical study.

RQ2: Does our approach can generate test data for any data-flow coverage criterion in a effcient way? According to the result reported in the Tables 4.3 and 4.4, we answer positively this question.

For Dijkstra, Table 4.3 reports all obtained results. Even though a significant total number of infeasible clear-paths (587) are completely proved, on average, less than 0.07s, is needed to cover a test target or to prove its infeasibility. Every def-use association either is covered or is proved infeasible. Thus, we can say that, with the existence of infeasibles paths, 100% coverage is reached for each data-flow criteria.

In less time, we also obtained the same percentage of coverage (100%) for Merge, Sample and Bsearch. Contrary to these units, for both tritype versions, the All-du-paths criterion does not reach 100% , because the test targets require to cover some infeasible paths that are impossible to prove without using a complete enumeration of the search space. In the case of tritype, all generated constraints are linear, so using a linear solver such as CPLEX (IBM, 2009a) may easily prove these infeasible paths, but this is not the case for any units.

Table 4.3: results of applying CPA-STDG on Dijkstra program to reach different data-flow coverage criteria (kPath=2)

Coverage Criterion	#Criterion Constraints	#Test Data Generated	#Infeasible Clear-path	#Def-use Covered	Time(s)
All-DU-Path	211	27	124	132	25.14
All-Use	211	15	124	87	12.07
All-p-Use-Some-c-Use	145	13	86	59	2.65
All-p-Use	136	13	84	52	1.96
All-c-Use-Some-p-Use	78	13	44	37	9.88
All-c-Use	75	13	40	35	8.67
All-Def	39	7	85	25	1.63
Total	**895**	**101**	**587**	**427**	**70.15**

Table 4.4: Results of applying CPA-STDG on different programs for All-DU-Paths coverage

UUT	#Criterion Constraints	#Test Data Generated	#Infeasible Clear-path	#Clear-paths Covered	Time(s)
Triangle	19	14	1+7	14+46	386.17
Tritype	21	9	1+0	9+24	64.21
Merge	36	13	4+0	13+34	0.07
Sample	40	13	3+0	13+38	0.28
Bsearch	28	3	6+10	3+9	0.03

RQ2: Does the proposed search strategy and checking process enhance or weake the performance of the proposed approach? For the UUTs Dijkstra, Tritype, Merge, and Sample, the solving process could not reuse any proved infeasible clear-paths to prune the search. This can be explained the reason that all infeasible clear-paths are independent. The set of covered clear-paths, which are detected at different check points, is five times greater than the number of test data generated. For Triange and Bsearch, the solving process could reuse proved infeasible clear-paths to prune the search many times. Only during the solving process of Triangle, one infeasible clear-path is proved was reused to detect seven other infeasible clear-paths. During the solving process of Bsearch, six infeasible clear-paths are proved were reused to detect ten other infeasible clear-paths. All proved infeasible or covered clear-paths are detected in one of the check points that are inserted in the three levels of search strategy. Also, the search strategy and checking process are an important factor that

increases the performance of the proposed approach. We thus answer yes for question RQ2.

4.3.2 Case Study 2: Control-flow Based Criteria

In this case study we compare CPA-STDG to two different approaches using two coverage criteria: CPA-STDG compared to (Gotlieb *et al.*, 2000) using the criterion all-branch coverage and CPA-STDG compared to (Williams *et al.*, 2005; Botella *et al.*, 2009) using the criterion all-path coverage. We based our comparisons on the time required to generate a test-data set for a each coverage criterion. All experiments were performed using ILOG OPL Studio 3.7.1, on a Windows XP, Intel Core 2 Duo 2GHz, 2GB of memory. For the first comparison, we limit the search time to 5 minutes.

Comparing Our Approach to a GOA

To compare CPA-STDG with the GOA proposed by Gotlieb *et al.* (2000), we translated the triangle program into two CSPs. In a first version $CSP1$, we used the approach proposed by Gotlieb *et al.* (2000) and in a second version $CSP2$, we use our approach. The *tri_type* function contains six conditional statements. Our goal is to achieve all-branch coverage. For each CSP, we created a set of twelve equivalent constraints, one per branch. Then, we solved each constraint in conjunction with CSP. Table 4.5 provides details of the results achieved when solving CSPs.

In the first seven branches, the time required to solve the $CSP1$ which is generated according to Gotlieb *et al.* (2000)'s approach is slightly better than the time required to solve ours $CSP2$, one reason being that their approach contains only program variables, whereas our approach also contains node variables, which requires an additional time during search. Another reason is that these seven branches are easy to reach (not-triangle, scalene). However, ours is more efficient in the last five branches, which are more complicated to achieve (equilateral, isosceles). After five minutes waiting for each branch, the $CSP1$ generated by Gotlieb *et al.* (2000)'s approach could not generate test-data to cover the last five branches that represent 40% of the program, whereas our approach generated a test datum for every branch.

Comparing Our Approach to a POA

To compare our approach with the POA (i.e., dynamic symbolic execution approach) proposed by Williams *et al.* (2005) we used our approach to translate all programs they used to evaluate their approach. Then we compared our results with those reported in (Williams

Table 4.5: Comparing all-branch coverage with Gotlieb et al.'s approach on *try_type* program.

Decision-node	Branch	**Time (s)**	
		$CSP1$	$CSP2$
D0	1	0.01	0.01
	-1	0.01	0.75
D1	1	0.77	0.95
	-1	0.01	0.75
D2	1	0.01	0.09
	-1	0.01	0.75
D3	1	0.01	0.01
	-1	>300	0.75
D4	1	>300	0.01
	-1	>300	0.75
D5	1	>300	0.01
	-1	>300	0.75

Table 4.6: Comparing all-path coverage with PathCrawler (Williams *et al.*, 2005) on *try_type*, *Sample*, and *Merge* programs

Program	**k-Path**	**#feasible Paths**	**Our Approach**		**PathCrawler (2005)**	
			#test data	**Time (s)**	**#test data**	**Time (s)**
try_type	-	10	10	0.001	14	0.010
Sample	3	240	240	0.060	241	0.270
Merge	2	17	17	0.080	19	-
	5	321	321	0.148	337	0.780
	10	20,481	20,481	28.640	20,993	116.000

et al., 2005) (Table 4.6) and with those recently published in (Botella *et al.*, 2009) (Table 4.7).

Merge program asks for two sorted arrays, the number of items to copy from each of them and returns a sorted array. This program contains three loops: the first one contains an if statement, the second and the third cannot be executed together in a same feasible path, so the number of feasible paths is $2 \cdot k \cdot 2^k + 1$, where k is the maximum allowed number of iterations. Note, that this formula is not valid for the results published in (Botella *et al.*, 2009) because they limited the maximum number of items to copy from each array by ten.

By definition, a test datum which is generated for all k-paths coverage should not exceed k iterations in a loop. The mechanism of test data generation based on partial path predicate (incomplete path predicate) used in PathCrawler can generate some test data that exceed this limit. These superfluous test data explain the difference in the number of test data with our approach. We generated the exact number of test data equivalent to the number

Table 4.7: Comparing all-path coverage with PathCrawler (Botella *et al.*, 2009) on *Merge* program.

k-Path	#feasible Paths	Our Approach		PathCrawler (2009)	
		#test data	Time (s)	#test data	Time (s)
2	17	17	0.080	19	0.330
5	321	321	0.148	337	0.800
10	12,287	12,287	18.163	12,798	37.200
15	204,931	204,931	827.250	216,371	876.350
All-Paths(19)	705,431	705,431	5,486.953	705,431	3,407.980

of feasible paths, while PathCrawler generated more test data (512 superfluous test data for $k = 10$), but they are not consistent with the chosen k: In (Williams *et al.*, 2005) (on page 8), the test set generated by PathCrawler for the value of $k = 2$ contains two test data (12 and 17) that are not valid for this value, because these test data are run the first loop in three iterations.

In terms of execution time our approach has proved its efficiency especially in the last two programs (Merge and Sample) where it is almost five times faster than PathCrawler. In our approach the time required for program translation was ignored because we did this task manually and this time is not significant comparing to the time required to solve the global CSP.

Generating a test data set to cover all paths without limiting the number of iterations shows that PathCrawler is slightly more efficient. For a value of k greater than ten, the number of iterations of the last two loops does not exceed ten. But our approach uses a single value for k, which means an additional number of infeasible paths to prove. For k less than or equal to ten our approach is clearly better than PathCrawler (Botella *et al.*, 2009). In practical terms, for a program that contains loops, the structural testing limits the number of iterations by a small number that is usually equal to two or three. Generally, the all-path coverage criterion without limitation on the number of iterations is not realistic (Myers, 1979). We also observe that for $k = 15$, PathCrawler generates a set of 11,440 extra test data that need extra time to apply.

4.3.3 Discussion

We also evaluated the performance of our novel approach in terms of required time to generate a set of test data that covers a given data-flow criterion. The results showed the importance of using constraint programming to generate test data, especially to prove the infeasibility of some infeasible clear-paths.

Yet, several threats potentially impact the validity of the results of our empirical study.

1. Threats to *construct validity* concern the relationship between theory and observation. In our study, these threats can be due to the fact that five UUTs are from the literature, even though previously-used to exemplify and study the structural testing (Collavizza et Rueher, 2006; Gotlieb *et al.*, 2000; McMinn, 2004; Williams *et al.*, 2005), and thus might represent real code. However, we used Dijkstra, which is a real program that is used in several systems. Finally, to show the level of testing difficulty, we computed the number of linearly-independent paths proposed by McCabe (1976). No UUT contains pointers, dynamic allocation , floating-point numbers, and function calls. In general, these are classical problems for constraint-based approaches and they are open research questions.

2. Threats to *internal validity* concern external factors that may affect an independent variable. We limited the number of loops iterations to 2 and the length of arrays in function of k-path. These limitations are almost invariably used in the literature. We also limited the domain size and the numbers of fails for tritype and triangle. We do not claim that our approach is able to prove any infeasible test objective, these two UUTs contain an example of infeasible path that our approach cannot prove so far. We also translated UUT using a semi-automated method, we meticulously realized this steps, and we took all necessary precautions to apply the approach as is. This step may not influence the behavior of the approach.

3. Threats to *external validity* involve the generalization of our results. We evaluated our novel approach on six UUTs to generate test data for seven data-flow criteria. First experiments are promising but, of course, to validate the proposed approach more experiments must be performed on larger units.

4. Threats to *conclusion validity* involve the relationship between the treatment and the outcome. To overcome this threat, we inspected manually the results obtained for two UUTs: Tritype and Triangle: we generated def-use associations manually and we compared them with the results returned by the implementation. For both UUTs, both generated use-def associations sets manual and automatic were the same. Finally, we created a program that use our approach to generate test data for all-paths coverage. It generated 14 data test for Triangle and 10 for Tritype, these numbers are exactly the well-known numbers of feasible paths in these two UUTs.

4.4 Summary

In this chapter, we introduced a new constraint-based approach for structural testing called CPA-STDG. Its novelty is in exploring execution paths by using CP's heuristics. The approach models a program with its CFG by a CSP to efficiently generate test data for many coverage criteria. It is the first approach that can answer to different coverage criteria using the same CSP model. We showed that modeling a program and its CFG by one CSP keeps the program's structural semantics and therefore it can generate test data for any structural coverage criteria or test target. We have already given, in Section 4.1, some ways in which our approach can be used to generate test data either for control-flow based criteria or data-flow based criteria.

We validated our novel approach on six programs and seven well-defined data-flow criteria: all combinations were performed, 42 experiments realized. The twelve most important of them are reported in this work, they showed the effectiveness and efficiency of the new approach. Thus, the results obtained from comparing CPA-STDG on different benchmarks to different approaches, either POA or GOA, are very promising. Despite the scalability issue, we can consider the obtained results as a very good starting point for using CPA-STDG to guide SB-STDG.

CHAPTER 5

COMBINING CB-STDG AND SB-STDG

Evolutionary testing uses a GA to generate test data. Traditionally, a random selection is used to generate an initial population and also, less often, during the evolution process. Such selection is likely to achieve lower coverage than a guided selection. Consequently, we define two novel concepts: (1) a Constrained Population Generator (CPG) that generates a diversified initial population that satisfies some test target constraints; and (2) a Constrained Evolution Operator (CEO) that evolves test candidates according to some constraints of the test target. Either the CPG or CEO may substantially increase the chance of reaching adequate coverage with less effort. In this Chapter, we use CPA-STDG to model a relaxed version of the UUT as a CSP. Based on this model and the test target, a CPG generates an initial population. Then, an evolutionary algorithm uses a CEO and this population to generate test input leading to the test target being covered. CSB-STDG combines CB-STDG and SB-STDG and overcomes the limitations associated with each of them. Using eToc, an open-source SB-STDG tool, we implement a prototype of this approach. We present the empirical results of applying both CPG or CEO on three open-source programs and show that CPG or CEO improve SB-STDG performance in terms of branch coverage by 11% while reducing computation time.

5.1 Motivating Example

We use the program in Fig. 5.1 as a running example. It considers the problem of generating a test input to reach Targets 1, 2, and 3. The three targets reflect different problems of test input generation.

- Target 1 is easy to reach;
- Target 2 is unreachable because $!(x \leq 0 || y \leq 0) \& !(x < y/2) \& (y > 3 \times x)$ is unsatisfiable;
- Target 3 is hard to reach because it contains nested predicates and involves the native function call, fun, that returns x^2/y.

CB-STDG can generate test inputs for Target 1 and prove that Target 2 is unreachable. However, if we suppose that the function fun cannot be handled by a particular constraint solver, a static CB-STDG approach, SE or GOA, cannot generate test inputs for Target 3. Thus, DSE can also fail to derive test inputs for Target 3 in a reasonable amount of time.

```
1  int foo(int X, int Y){
2    if(X<=0 || Y<=0)
3      return 0;
4    int Z;
5    if ((X < Y/2)|| (Y==0))
6      Z= 1; //Target 1
7    else if (Y>3*X)
8      Z=2; //Target 2
9    else{
10     Z = fun(X,Y);
11     if ((Z >8) && (Y==10))
12       if(Z==Y)
13         Z=3;//Target 3
14   }
15   return Z;
16 }
```

Figure 5.1: Motivating example for combining CB-STDG and SB-STDG

SB-STDG can derive test inputs for Target 1 but it takes more time than CB-STDG. Therefore, if the search space is large, SB-STDG may take a very long time before generating a test input for Target 3 (it needs $x = 10$ and $y = 10$). For Target 2, SB-STDG may search forever without proving its infeasibility.

Target 3 is problematic for both approaches: it is a nested branch predicate McMinn (2004) for SB-STDG and it contains an unsupported function for CB-STDG. However a hybrid approach can take advantage of both to generate test input for Targets 1 and 3, and it may prove the unreachability of Target 2. We confirmed this fact by generating a test input to reach Target 3 using all three approaches: SB-STDG (eToc Tonella (2004) and a hill climbing implementation), CPA-STDG Sakti *et al.* (2011), and a combination of these approaches. We concluded that eToc and Hill Climbing were unable to generate test inputs to reach Target 3 after 45000 fitness calculations in domain $[-20000, 20000]$. As well, CPA-STDG was unable to reach Target 3. However, a hybridization eToc+CPA-STDG (resp. HC+CPA-STDG) was able to generate test input just after 200 (resp. 600) fitness calculations.

The key idea of our hybridization is to proceed in two phases. First, generating a relaxed version of foo by replacing the function fun with an uninitialized variable typed as foo's return value. Then CPA-STDG uses the relaxed version to generate pseudo test inputs. In a second phase, SB-STDG uses those pseudo test inputs as candidates to generate the actual test inputs.

Now, consider for example the program in Fig.5.1. In order to reach Target 3, the conjunction of the negation of the first three conditional statements and the last two conditional statements must be fulfilled. For the last two conditions, z depends on $fun(x, y)$'s return value. Using CPA-STDG Sakti *et al.* (2011) the path condition can be written as follow:

$not((x_0 \leq 0)||(y_0 \leq 0)) \wedge not((x_0 < y_0/2)||(y_0 = 0)) \wedge not(y_0 > 3 \times x_0) \wedge z_3 = fun(x_0, y_0) \wedge ((z_3 > 8) \wedge (y_0 = 10)) \wedge (z_3 = y_0)$.

The relaxed version is obtained by ignoring the constraint $z_3 = fun(x_0, y_0)$. Then, the path condition becomes $not((x_0 \leq 0)||(y_0 \leq 0)) \wedge not((x_0 < y_0/2)||(y_0 = 0)) \wedge not(y_0 > 3 \times x_0) \wedge ((z_3 > 8) \wedge (y_0 = 10)) \wedge (z_3 = y)$. As the relaxed path condition is less restrictive than the actual one, generated pseudo test inputs won't necessarily trigger Target 3, but they satisfy a big part of the path condition. By solving the relaxed constraint, we obtain solutions of the form ($x_0 \geq 5$, $y_0 = 10$). Therefore, the search space size has been reduced. Using those pseudo test inputs as an initial population for an evolutionary algorithm can reduce significantly the effort needed to generate test data.

In this example, only one path led to Target 3. If there are many paths that can reach the test target, generating the test data that allows covering different paths or different branches may assure a diversified population for SB-STDG.

5.2 CSB-STDG

The assumption underlying the research work presented in this Chapter supposes that using a diversified initial population that partly satisfies the predicates leading to the test target can reduce significantly the effort required to reach this test target. Our approach is called CSB-STDG because it starts, in a first phase, by using CPA-STDG to generate an initial population and then, in a second phase, it uses SB-STDG to generate test inputs. We propose to replace the random generation of an initial population used by SB-STDG with a set of pseudo test input generated using a CPA-STDG approach.

Fig. 5.2 presents the whole inputs search space of a program P: the parts A, B, C, D, and E are the CPA-STDG solutions space of a relaxed version of P which are called *pseudo test inputs*, while stars are actual test inputs. This example shows that a random test input candidate is likely to be too far from an actual test inputs compared to some pseudo test inputs. The parts C and E don't contain any test input, so a pseudo input from these two parts may also be far from an actual test input. A population that takes its candidates from different parts is likely to be near of a test input, we call it a *diversified population*. We can offer an acceptable level of diversity by generating a pseudo test input for every consistent branch with the test target (all-branch), and a high level of diversity by generating a pseudo test input for every path leads to the test target (all-path).

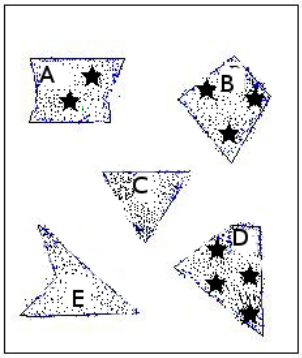

Figure 5.2: Inputs search space.

5.2.1 Unit Under Test Relaxation

To avoid traditional CPA-STDG problems, we introduce a preliminary phase called program relaxation. We propose to apply CPA-STDG on a relaxed version of the UUT. A relaxed UUT version is a simplified version, of the original one, that contains only expressions supported by the constraint solver: the expressions that generate constraints whose consistency can be checked by the constraint solvers are kept in the relaxed version, while all the other expression are relaxed or ignored, e.g., for an integer solver, expressions that can generate constraints over string or float are ignored, unsupported operators (expressions) are relaxed, and a native function call that returns an integer is relaxed.

A relaxed version is obtained by applying the following rules:

- Any unsupported expression (function call, operator) is relaxed: the expression is replaced by a new variable. Fig. 5.4 shows a relaxed version of $intStr$, which is shown in Fig.5.3, for an integer solver. We suppose that the solver cannot handle the shift operator, so the expression that uses this operator has been replaced by a new variable $R1$. The variable $R1$ is not initialized. Therefore, when we will translate the relaxed version into a CSP, the CSP variable $R1$ can take any integer value.

- Any statement over unsupported data type is ignored. In Fig. 5.4, The relaxed version of $intStr$ ignores the statement $String\ s = S1 + S2$; and the condition $s.equals(\text{“}OK\text{”})$ because those two expressions are over strings.

- For each data type that needs a different solver a new relaxed version is created. The function $intStr$ needs integer and string type as inputs. If we have an available solver for string, then we generate another relaxed version over string. Fig. 5.5 shows a relaxed version of $intStr$ for a string solver.

```
1  intStr(int X,int Y,
2   String S1, String S2){
3    int y= X<<Y;
4    int x=y+X/Y;
5    String s=S1+S2;
6    if((s.equals("OK")
7      && x>0)
8      && s.length()>x)
9       return 1; //Target
10   return 0;
11 }
```

Figure 5.3: intStr function

```
1  intStr(int X,int Y){
2    int R1,R2;
3    int y= R1;
4    int x=y+X/Y;
5    if(
6      x>0)
7      && R2>x)
8       return 1; //Tar
9    return 0;
10 }
```

Figure 5.4: Relaxed version for an integer solver

```
1  intStr(String S1,
2       String S2){
3
4    String s=S1+S2;
5    if((s.equals( 'OK")
6      )
7      )
8       return 1; //Tar
9    return 0;
10 }
```

Figure 5.5: Relaxed version for a string solver

- For loops, CPA-STDG cannot model an unlimited number of iterations. In general a constant k-path (equal 1, 2, or 3) is used to limit the number of loop iterations. We can model a loop in two different ways: first, we force a loop to stop at most after k-path iterations, in this case some feasible paths may become infeasible; second, we don't force a loop to stop and we model just k-path iterations, after which the value of a variable assigned inside a loop is unknown. We use the second case and we relax any variable assigned inside a loop just after this loop. The variable is assigned a new uninitialized variable.

5.2.2 Collaboration between CPA-STDG and SB-STDG

Every test input generated is a result of a collaborative task between CPA-STDG and SB-STDG: CPA-STDG generates a pseudo test input, and then SB-STDG generates an actual test input. Our main contribution here is to define the information exchanged and connection points that can make CPA-STDG useful for SB-STDG. SB-STDG needs new test input candidates at three different points:

1. During the generation of the initial population;
2. When it restarts, i.e., when it reaches the attempt limit of evolving;
3. During its evolving procedure.

For the first and the second points, a CPA-STDG can generate the whole or a part of the population. If CPA-STDG is unable to generate the required number (population size) of candidates the usual generation procedure (random) can be called to complete the population. We called this technique CPG. For the third point, CPA-STDG can participate to guide the population evolution by discarding candidates that break the relaxed model and only allowing

candidates that satisfy the relaxed model, we called this technique CEO. But frequent calls to CPA-STDG seem to be too expensive in practice, this can weaken the main advantage of evolutionary algorithms which is their speed. Therefore, we propose to limit the number of CPA-STDG calls during the population evolving procedure.

5.3 Implementation

We have implemented an integer version (with an integer solver) of our approach CSB-STDG by using a new implementation of our CPA-STDG Sakti *et al.* (2011) and by extending eToc Tonella (2004). Fig. 5.6 shows the overview of the implementation, built over six components: Program instrumentor, Constraint models Generator, CPA-STDG Core, a CEO, a CPG, and SB-STDG Core. The main components are CPA-STDG Core and SB-STDG Core. These two components communicate via shared target and population. They identify sub-targets (branches) in the same way by using the Program Instrumentor component as a common preprocessing phase.

In the architecture, the SB-STDG Core acts as the master and CPA-STDG Core plays the role of slave. When SB-STDG Core needs test input candidates, it requests CPA-STDG Core by sending its current test target. To answer the request, CPA-STDG Core uses CEO or CPG and replies by sending a pseudo test input, proving the inaccessibility of the target, or by simply saying that the execution time limit is reached. Then SB-STDG evolves its new population. This process is repeated until the test target is covered or some condition limit is met.

5.3.1 SB-STDG Core

We use eToc that implements a genetic algorithm to generate test cases for object oriented testing. eToc begins with instrumenting the UUT to identify the test target and to keep trace of the program execution. eToc generates a random initial population whose individuals are a sequence of methods. eToc follows the GA principle: to evolve its population it uses a crossover operator and four mutation operators. The main mutation operator that interests us mutates method arguments. To adapt eToc to our requirements, we modified the argument values generator from a random generator to a CPG. Thus, we modified one mutation operator to make it a CEO.

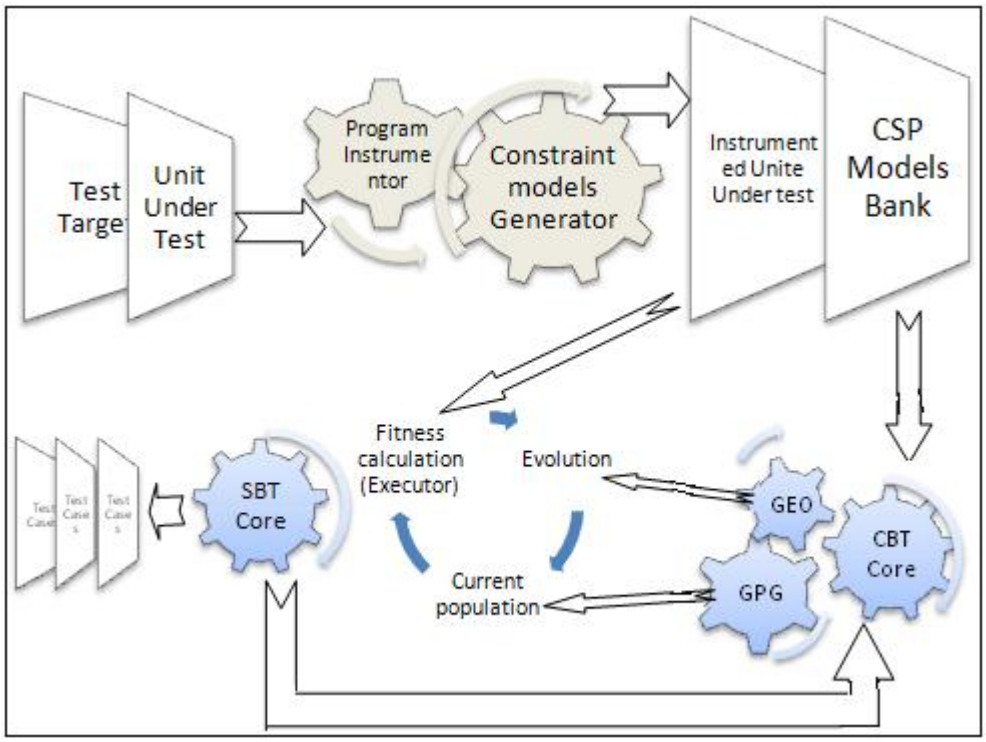

Figure 5.6: Implementation overview

5.3.2 CPA-STDG Core

Constraint Models Generator.

This component takes a UUT as input optionally instrumented with eToc program instrumentor. For each Java class method a specific structure of control flow graph is generated. In addition, using the Choco[1] language a constraint model for a relaxed version of this method is generated. All generated models constitute a CSP Models Bank that is used by CPA-STDG Core during test input generation.

Constrained Evolution Operator (CEO)

CEO can be implemented as crossover operator, mutation operator, or neighbourhood generator. In this work, we implemented CEO as a mutation operator. With a predefined likelihood the genetic algorithm calls CEO by sending the methods under test, the current test target, current parameters values, and the parameter to change. CEO uses this information to choose the adequate CSP model and to fix the test target and parameters except

[1]Choco is an open source java constraint programming library. urlhttp://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf

those required to change. After that the model is solved and a new value is assigned to the requested parameter. If the solver does not return a solution then all parameters are assigned arbitrary values.

Constrained Population Generator (CPG)

When the eToc genetic algorithm starts or restarts, randomly it generates its chromosomes (methods sequences), and then it calls CPG to generate parameters values. For each function in the population, this generator check a queue of pseudo test inputs, if there is a pseudo test input for this method, then the method's parameters are assigned and this pseudo input test is deleted. Otherwise the CPG generates a pseudo test input for each test target not yet covered in this method. One of these pseudo test inputs is immediately assigned to this method's parameters, the rest are pushed into the queue. During solving test target if a target is proved infeasible, then the generator drops it from the set of target to cover.

5.4 Empirical Study

The *goal* of our empirical study is to compare our proposed approach against previous work and identify the best variant among the two proposed techniques in Section 5.2.2 and a combination thereof. The *quality focus* is the performance of the CPG technique, the CEO technique, CPG+CEO, CPA-STDG, and SB-STDG to generate test inputs set that covers all-branches. The *context* of our research includes three case studies: Integer, BitSet, and ArithmeticUtils were taken from the *Java* standard library and *Apache Commons project*[2]. BitSet was previously used in evaluating white-box test generation tools Tonella (2004); Inkumsah et Xie (2007). These classes have around 1000 lines of java code. The SB-STDG used (eToc) was not able to manage array structures and long type. We fixed the long type limitation by using *int* type instead, but we had to avoid testing methods over array structures. Therefore, we tested the whole *BitSet* and *ArithmticUtils*, but only a part from *Integer* because it contains array structure an input data for some methods. The number of all branches is 285: 38 in Integer, 145 in BitSet, and 102 in ArithméticUtils. These classes were chosen specifically because they contain function calls, loops, nested predicates, and complex arithmetic operations over integers.

5.4.1 Research Questions

This case study aims at answering the following three research questions:

[2]The Apache Software Foundation. `http://commons.apache.org`

RQ1: Can our approach CSB-STDG boost the SB-STDG performance in terms of branch coverage and runtime? This question shows the applicability and the usefulness of our approach.

RQ2: When and at what order of magnitude is using our approach useful for SB-STDG? This question shows the effectiveness of our approach.

RQ3: Which of the three proposed techniques is best suited to generate test inputs in an efficient way?

5.4.2 Parameters

As shown in the running example, during the empirical study we observed that the difference, in terms of fitness calculations, between CSB-STDG and SB-STDG is very large. We think that comparing approaches using this metric is unfair because CSB-STDG uses CPA-STDG to reduce the number of fitness calculations. So to provide a fair comparison across the five approaches (CPA-STDG, SB-STDG, CPG, CEO, CPG+CEO), we measure the cumulative branch coverage achieved by these approaches every 10 seconds for a sufficient period of *runtime* (300 s). This period was empirically determined as sufficient for all approaches. To reduce the random aspect in the observed values, we repeated each computation 10 times. We think that the default integer domain in eToc $[-100, 100]$ is unrealistically small so to get a meaningful empirical study, we chose to fix the domain for all the input variables to a larger domain $[-2 \times 10^4, 2 \times 10^4]$. We kept the rest of eToc's default parameters as is. We used identical parameters values for all techniques.

For CPA-STDG, the solver uses the minDom variable selection heuristic on the CSP variables that represent CFG nodes as a first goal and on CSP variables that represent parameters as a second goal. The variable value is selected randomly from the reduced domain. To make the study scalable, we restrict the solver runtime per test target to 500 ms. This avoids the solver hanging or consuming a large amount of time.

5.4.3 Analysis

Fig. 5.7 summarizes the branch coverage percentage in terms of execution time for the BitSet class. Overall, the CPG technique is the most effective, achieving 89.5% branch coverage in less than 40 s. Also, CEO and CPG+CEO reach 89.5% but after 70 s. eToc was unable to go beyond 85.78%, which is attained after 120 s. CPA-STDG performs badly, its best attained coverage is 78.94%. This figure shows that the proposed three techniques can improve the efficiency of eToc in terms of percentage coverage and execution time.

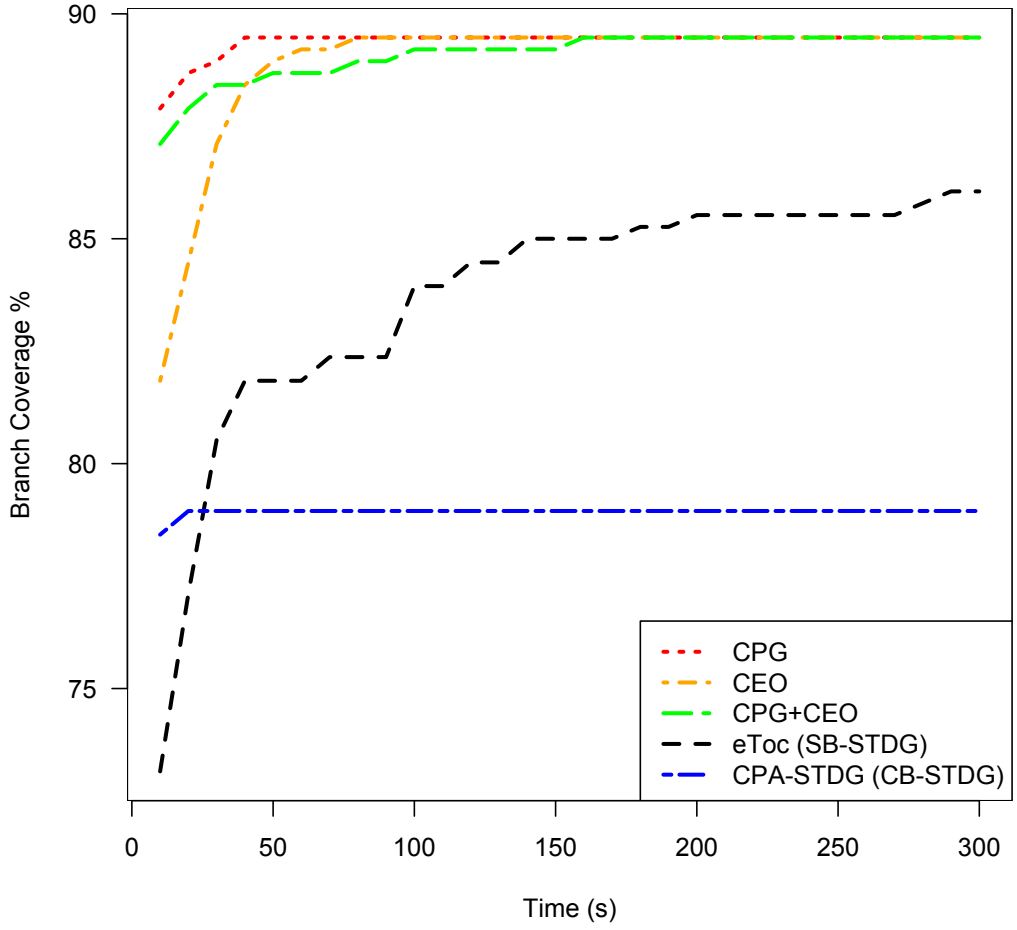

Figure 5.7: Comparing all techniques on Integer.java

We analysed branch coverage percentage in terms of execution time on BitSet and on ArithmeticUtils. We got two graphics that resemble Fig. 5.7 with a slight difference: eToc starts better than the three proposed techniques for the first twenty seconds, but CPG rapidly makes up for lost time outperforming eToc just after 20 s. Also CEO and CPG+CEO outperformed eToc, but they needed a little more time especially on ArithmeticUtils. CPA-STDG performs always worse than even the weakest proposed techniques.

Finally, Fig. 5.8 reflects the achieved results for all classes Integer, BitSet, and ArithmeticUtils. It confirms that the CPG technique is the most effective, and eToc starts better than the proposed techniques during the first twenty seconds.

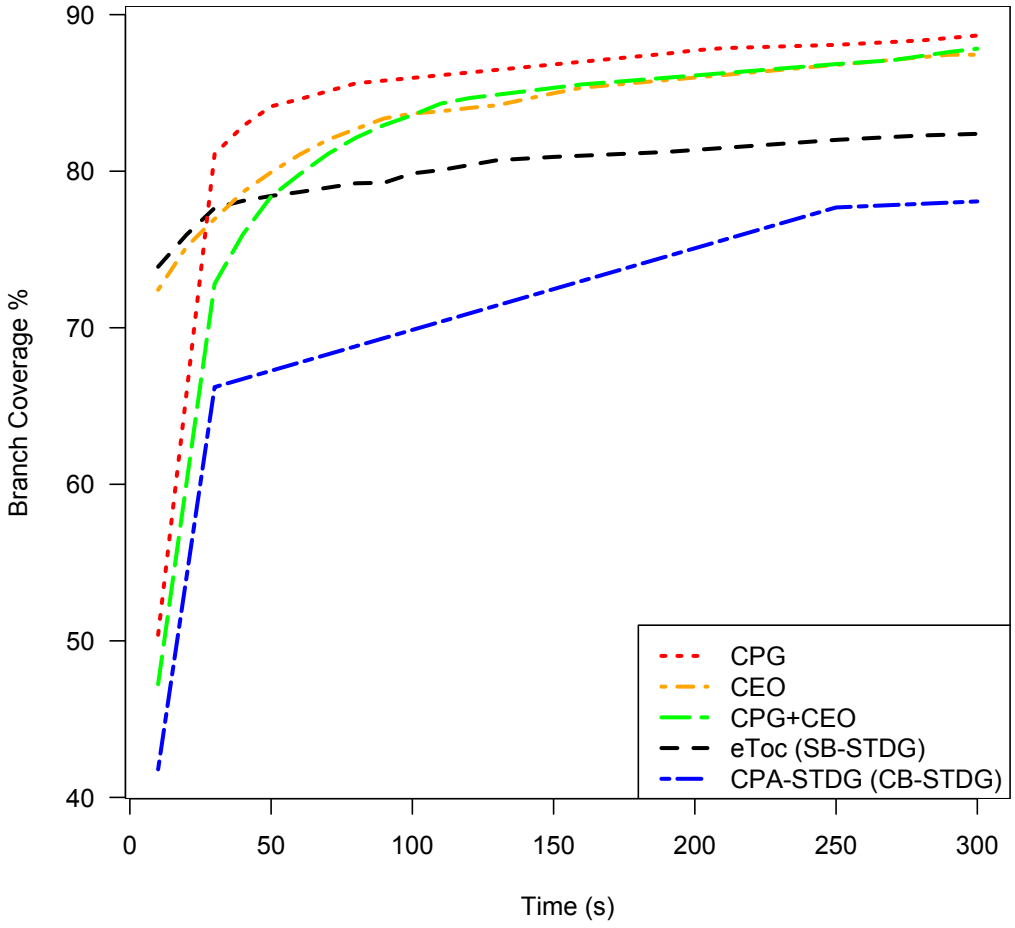

Figure 5.8: Comparing all techniques on all the three java classes

5.4.4 Study Discussions

For the given classes, it is clear that CPG outperforms all techniques in terms of execution time and branch coverage. Also CEO and CPG+CEO perform better than eToc. Therefore, the proposed techniques boost SB-STDG implemented in eToc. This result may be due to the kind of UUTs tried, which essentially use integer data types. More evidence is needed to verify whether the advantage of the proposed techniques represents a general trend. Yet, on the selected UUTs and the tool eToc **we answer RQ1 by claiming that the CSB-STDG techniques can boost the SB-STDG**.

Over almost all graphics, we observed that eToc starts better than the proposed tech-

niques. Therefore the latter are not useful for the first twenty seconds. This behaviour is due to the frequency of solver calls: at the start time all targets are not yet covered even the easiest one which can be covered randomly. Therefore, a combination that uses only SB-STDG for a small lapse of time or until a certain number of fitness calculations and then uses CSB-STDG, may perform better. Also, we observed that after this time the CSB-STDG techniques quickly reach a high level of coverage. This is because at this moment CSB-STDG takes advantage of both approaches: it includes branches which are covered either by SB-STDG, CPA-STDG or by their combination. Thus, **we answer RQ2 by claiming that the CSB-STDG is more useful after the starting time**.

Even though CEO and CPG+CEO outperform SB-STDG, we think that they don't perform as expected. On BitSet class these two techniques take a significant time before beating SB-STDG. There are several factors that can influence the performance of the CEO. First, the frequency of solver calls is very high; it makes a call for every mutation. Second, in object oriented testing, a method under test does not necessarily contain the test target — this can make the mutation operator useless. Third, the mutation that we used imposes to fix part of the parameters which can make the CSP infeasible or hard to solve. These three factors are the main sources of CEO weakness. CPG+CEO is indirectly influenced by these factors by using CEO.

The CPG technique enhanced eToc by an average of 6.88% on all classes: 3.60% on Integer, 5.65% on BitSet, and 11.37% on ArithmeticUtils. These values did not take into account the proved infeasible branches: just on ArithmeticUtils CPG has proved 4 infeasible branches. We confirmed manually that these branches were infeasible because they use in their predicates some values out of the domain used. According to Inkumsah et Xie (2007) the branches not covered by eToc are very difficult to cover. Therefore, a percentage ranging between 3.6% and 11.37% is a good performance for CPG. Thus, **we answer RQ3 by claiming that CPG is more useful to generate test inputs**.

5.4.5 Threats to Validity

The results showed the importance of using CSB-STDG to generate test data, especially to improve the SB-STDG performance in terms of runtime and coverage.

Yet, several threats potentially impact the validity of the results of our empirical study. A potential source of bias comes from the natural behaviour of any search based approach: the random aspect in the observed values. This can influence the internal validity of the experiments. In general, to overcome this problem, the approach should be applied multiple times on samples with a reasonable size. In our empirical study, each experiment took 300 seconds and was repeated 10 times. The coverage was traced every 10 s. The observed values

become stable after 120 s. Each tested class contains around 1,000 LOCs and more than 100 branches. Therefore, experiments provided a reasonable size of data from which we can draw some conclusions, but more experiments are strongly recommended to confirm or refute such conclusions.

Another source of bias comes from the eToc genetic algorithm parameters: we didn't try different combinations of parameters values to show empirically that the approach is robust to eToc parameters. This can affect the internal validity of our empirical study.

Another potential source of bias includes the selection of the classes used in the empirical study, which could potentially affect its external validity. The BitSet class has been used to evaluate different structural testing approaches before Tonella (2004); Inkumsah et Xie (2007), thus it is a good candidate for comparing the different proposed techniques. The two other classes were chosen because they feature integers and because they contain common problems in CPA-STDG and SB-STDG (e.g, path that contains nested predicates and native function calls) and they represent widely used classes with non-trivial sizes.

5.5 Summary

In this Chapter, we presented a novel combination of SB-STDG and CPA-STDG to generate test inputs, called CSB-STDG. The novelties of our approach lie in its use of a relaxed version of a UUT with CPA-STDG and of CPA-STDG solutions (pseudo test inputs) as test input candidates in service of SB-STDG. We identified three main points where CPA-STDG can be useful for SB-STDG. For each point we proposed a technique of combination: a CPG that uses CPA-STDG to generate test input candidates for SB-STDG; and a CEO that uses CPA-STDG to evolve test input candidates. We implemented a prototype of this approach. Then we compared three variants of CSB-STDG with CPA-STDG and SB-STDG. Results of this comparison showed that CPG outperforms all techniques in terms of runtime and branch coverage. It is able to reach 89.5% branch coverage in less than 40 s. Also CEO and CPG+CEO perform better than SB-STDG in terms of branch coverage. The obtained results are promising but more experiments must be performed using different sort of solvers (String, floating point) to confirm if the absolute advantage of the proposed techniques represents a general trend.

CHAPTER 6

CONSTRAINT BASED FITNESS FUNCTION

One major limitation of SB-STDG is an insufficiently informed fitness function to guide search toward a test target within nested branches. To address this problem we propose CB-FF, a fitness function based on concepts well-known to the CP community, such as constrainedness and arity, to evaluate test-data candidates. It statically analyzes the test target to collect information about branches leading to it. Then it defines a penalty value for breaking each branch according to its arity and its constrainedness. Branch penalties are used to determine how close a test-data candidate is to reach a test target.

We empirically evaluate the proposed fitness functions on different benchmarks to compare the number of evaluations required to generate test data with the state-of-the-art of fitness functions, i.e., Approach Level (Wegener *et al.*, 2001) and Symbolic Enhanced Fitness Function (Baars *et al.*, 2011). Preliminary experiments promise efficiency and effectiveness for the new fitness functions.

6.1 Motivation

To generate test data, SB-STDG uses a meta-heuristic guided by a fitness function. The *approximation-level* (Wegener *et al.*, 2001) and *branch-distance* (Tracey *et al.*, 1998) measures are largely used for computing the fitness function value (McMinn, 2004). These measures are respectively the number of branches leading to the test target that were not executed by the test candidate and the minimum required distance to satisfy the branch condition where the execution of the test candidate has left the target's path (that branch is called the *critical branch*).

Assume that our test target is reaching the statement at line 5 in the code fragment at Fig. 6.1. The branch distance of a test candidate (a, b, c) is equal to $|b - c|$, from the first branch. If this distance is equal to 0 then the first branch is satisfied and the branch distance is computed according to the second branch $(y > 0)$ which is equal to $\max(0, 1-b)$, and so on. The approach level is an enhancement of the branch distance fitness that links each branch to a level. The level of a branch is equal to the number of branches that separate it from the test target. In Fig. 6.1 the branch at line 4 has level 1 because there is only one branch to satisfy to reach the test target. The branch at line 3 has level 2 because there are two branches to satisfy, and so on. The approach level fitness function (f_{AL}) for a test candidate i and a

```
1   sample(int x,int y,int z){
2     if(y==z)
3       if(y>0)
4         if(x==10)
5           ...//Target
6   }
```

Figure 6.1: Code fragment

branch b is $f_{AL}(i) = level(b) + \eta(i, b)$, where b is the critical branch of i. The second term η is a normalized branch distance. Among the many methods to compute it (Arcuri, 2010), we use $\frac{\delta}{\delta+1}$ where δ is the branch distance. This fitness function does not take into account non-executed branches. SB-STDG is a dynamic approach: it executes the program under test and observes its behavior. Therefore it only analyses executed branches. Suppose that we have two test candidates $i_1 : (10, -30, 60)$ and $i_2 : (30, -20, -20)$ and we need to choose one of them. The approach level fitness function chooses i_2 ($f_{AL}(i_2) = 2 + \frac{21}{22} = 2.9545$) because it cannot see that i_1 ($f_{AL}(i_1) = 3 + \frac{90}{91} = 3.9890$) satisfies the branch at line 4. A careful static analysis would detect that i_2 needs more effort than i_1 since it needs to change every value whereas i_1 could reach the target with a single change of the second value. Furthermore the likelihood of randomly choosing the value 10 for x in a large domain is almost null.

Recently, Baars *et al.* (2011) proposed the Symbolically Enhanced Fitness Function (f_{SE}) that complements SB-STDG with a simple static analysis, i.e., symbolic execution: $f_{SE}(i) = \sum_{b \in P} \eta(i, b)$ whether branch b is executed or not. For our example their fitness function also chooses i_2 ($f_{SE}(i_2) = 0 + \frac{21}{22} + \frac{20}{21} = 1.9068$) over i_1 ($f_{SE}(i_1) = \frac{90}{91} + \frac{31}{32} + 0 = 1.9577$) because it does not prioritize branch "$(x == 10)$" — all branches are considered equal.

In summary: (normalized) branch distance is accurate to compare test candidates on the same branch but can be misleading when comparing on different branches; approach level corrects this by considering the critical branch's rank on the path to the target but ignores non-executed branches; symbolic enhanced, a hybrid approach that complements the dynamic approach with static analysis, includes all branches on the path but does not consider their rank or any other corrective adjustment. We propose branch hardness as such an adjustment, complementing branch distance.

6.2 CB-FF Principle

We claim that test candidates that satisfy "hard" branches are more promising than those that only satisfy "easier" branches. Therefore we consider the difficulty to satisfy a constraint (i.e. a predicate used in a conditional statement or branch) as key information to measure

the relevance of a test candidate. We propose a *branch-hardness fitness function* to guide a SB-STDG approach toward test targets: it prioritizes branches according to how hard it is to satisfy them. As in the Symbolic Enhanced approach, we combine SB-STDG with static analysis of the non-executed branches, but additionally apply a hardness measure to every branch.

Here the difficulty to satisfy a constraint c is linked to its *arity* and its *projection tightness* (Pesant, 2005). The arity of a constraint c is the number of variables involved in c. For example, in Figure 6.1 the arity of each branch is as following:

- $arity(\text{“}y == x\text{”}) = arity(\text{“}!(y == x)\text{”}) = 2$;
- $arity(\text{“}y > 0\text{”}) = arity(\text{“}!(y > 0)\text{”}) = 1$;
- $arity(\text{“}x == 10\text{”}) = arity(\text{“}!(x == 10)\text{”}) = 1$.

The lower the arity of the constraint, the less freedom we have to choose some of its variables in order to modify the test-datum candidate. Correspondingly as a first indicator of hardness we define the parameter $\alpha(c) = \frac{1}{arity(c)}$ that ranges between 0 and 1.

The projection tightness is the ratio of the (approximate) number of solutions of a constraint to the size of its search space (i.e. the Cartesian product of the domains of the variables involved in c), in a way measuring the tightness of the projection of the constraint onto the individual variables. For example, in Figure 6.1 let's suppose that all domains (D) are equal to $[-99, 100]$, then the search space size, the approximate number of solutions, and the projection tightness of each branch are as following:

- The search space size
  - $search\_space\_size(\text{“}y == x\text{”}) = search\_space\_size(\text{“}!(y == x)\text{”}) = 200^2$;
  - $search\_space\_size(\text{“}y > 0\text{”}) = search\_space\_size(\text{“}!(y > 0)\text{”}) = 200$;
  - $search\_space\_size(\text{“}x == 10\text{”}) = search\_space\_size(\text{“}!(x == 10)\text{”}) = 200$.
- The approximate number of solutions (Pesant, 2005)
  - $approximate\_number\_of\_solutions(\text{“}y == x\text{”}) = 200$;
  - $approximate\_number\_of\_solutions(\text{“}!(y == x)\text{”}) = |D(y)|\dot{|}D(x)|-|D(y)\cap D(x)| = 200^2 - 200$;
  - $approximate\_number\_of\_solutions(\text{“}y > 0\text{”}) = 100$;
  - $approximate\_number\_of\_solutions(\text{“}!(y > 0)\text{”}) = 101$;

- $approximate\_number\_of\_solutions(\text{“}x == 10\text{”}) = 1$;
- $approximate\_number\_of\_solutions(\text{“!}(x == 10)\text{”}) = 199$.

- The projection tightness
  - $projection\_tightness(\text{“}y == x\text{”}) = \frac{200}{200^2}$;
  - $projection\_tightness(\text{“!}(y == x)\text{”}) = \frac{39800}{200^2}$;
  - $projection\_tightness(\text{“}y > 0\text{”}) = \frac{100}{200}$;
  - $projection\_tightness(\text{“!}(y > 0)\text{”}) = \frac{101}{200}$;
  - $projection\_tightness(\text{“}x == 10\text{”}) = \frac{1}{200}$;
  - $projection\_tightness(\text{“!}(x == 10)\text{”}) = \frac{199}{200}$.

A projection tightness close to 0 will indicate high constrainedness and hardness to satisfy a constraint. Correspondingly as a second indicator of hardness we define the parameter $\beta(c) = 1 - projection\_tightness(c)$ that ranges between 0 and 1.

In the following subsections, based on the two indicators of hardness we define two metrics Difficulty Coefficient and Difficulty Level to complete the branch distance and use them to express two fitness functions. These two metrics are designed to be used within meta-heuristic search to solve an test-data generation problem.

6.2.1 Difficulty Coefficient (DC)

The DC is a real number greater than 1. It is a possible representation of the hardness of a branch. Each constraint has its own DC that is determined according to its arity and tightness. DC is calculated by the following formula,

$$DC(c) = B^2 \cdot \alpha(c) + B \cdot \beta(c) + 1,$$

where $B > 1$ is an amplification parameter. In this work we use $B = 10$.

In the expression of $DC(c)$, we favor the indicator $\alpha(c)$ over the indicator $\beta(c)$ because we want to direct the search to keep solution candidates that satisfy constraints involving a small number of variables, then we aim to change other variables that may evolve current solution candidates to satisfy other constraints with a bigger arity.

To express a fitness function based on DC we use DC as a penalty coefficient for breaking a constraint. The key idea behind this fitness function is determining a *standard-branch-distance*, then sorting candidates according to their total standard-branch-distance.

To compute a fitness value for a test candidate i on a set of branches (constraints) C we apply the following formula:

$$f_{DC}(i, C) = \sum_{c \in C} DC(c) \cdot \eta(i, c).$$

This fitness function penalizes a broken constraint in a relative manner, i.e., it determines the penalty according to the DC and the normalized distance branch. In this way we may prefer a candidate that satisfies a hard branch but breaks an easier one with a large normalized distance, over another candidate that breaks the former with a small normalized distance but satisfies the latter.

Now we come back to our example at Fig. 6.1 and apply this new fitness function. Assume that all domains are equal to $[-99, 100]$. We compute DC for each branch as follows:

1. $DC(\text{“}y == z\text{”}) = 10^2 \cdot 0.5 + 10 \cdot 0.995 + 1 = 60.95$;
2. $DC(\text{“}y > 0\text{”}) = 10^2 \cdot 1 + 10 \cdot 0.5 + 1 = 106$;
3. $DC(\text{“}x == 10\text{”}) = 10^2 \cdot 1 + 10 \cdot 0.995 + 1 = 110.95$.

Then we use these DC values to compute a fitness value for each test candidate: $f_{DC}(i_1, C) = 60.95 \cdot \frac{90}{91} + 106 \cdot \frac{31}{32} + 110.95 \cdot \frac{0}{1} = 162.9677$; $f_{DC}(i_2, C) = 60.95 \cdot \frac{0}{1} + 106 \cdot \frac{21}{22} + 110.95 \cdot \frac{20}{21} = 206.8485$. Contrary to f_{AL} and f_{SE} this fitness function makes the adequate choice by choosing the test candidate i_1 instead of i_2. This type of decision may make f_{DC} more efficient than f_{AL} and f_{SE}.

6.2.2 Difficulty Level (DL).

DL is a representation of a relative hardness level of a constraint in a set of constraints. DL determines a constant penalty of breaking a constraint in a set of constraints to satisfy. We make the satisfaction of a hard constraint more important than satisfying all constraints at a lesser level. To favor a constraint over all other easier constraints, its DL must to be greater than the sum of DL values for all the constraints with a lower DC score. Therefore, we define the DL as

$$DL(c, C) = \begin{cases} |C|, & \text{if } r = 0 \\ 2^{r-1} \cdot (|C| + 1), & \text{if } r > 0 \end{cases}$$

where r is the rank (starting at 0) of c in the constraints set C in ascending order of DC.

To get a fitness value based on DL for a test candidate i on a set of branches C we apply the following formula:

$$f_{DL}(i,C) = \sum_{c \in C} \ell(i,c) + \eta(i,c), \text{ where } \ell(i,c) = \begin{cases} 0, & \text{if } \eta(i,c) = 0 \\ DL(c,C), & \text{if } \eta(i,c) \neq 0 \end{cases}$$

This fitness function allows to compare test candidates on different levels. It absolutely favors test candidates that have a smaller penalty value, i.e, test candidates that satisfy hard constraints. Contrary to the fitness function based on DC this one always prefers a test candidate that satisfies a hard constraint and breaks all easier constraints over a test candidate that breaks the hard constraint, even though it satisfies all the easier constraints.

6.3 Empirical Study

Our aim in this empirical study is to analyze the impact of our proposed fitness functions on SB-STDG in terms of effectiveness and efficiency: SB-STDG is considered more efficient if the new fitness functions are able to reduce the number of evaluations; it is considered more effective if the new fitness functions are able to cover more targets. We compare our proposal to the state of the art, f_{AL}, f_{SE}, and also to a natural combination of the two that applies the branch level as a corrective coefficient to each term of f_{SE}, which we denote f_{SEL}.

To perform our empirical study, we select two widely used meta-heuristic algorithms: Simulated Annealing (SA) and Evolutionary Algorithm (EA). To implement them we use the open source library opt4j (Lukasiewycz *et al.*, 2011). To define an optimization problem in opt4j, the user needs only to define a fitness function and a representation of a test candidate which, in our case, is a vector of integers. We defined a Java class that exports a fitness evaluator for each fitness function: f_{AL}; f_{SE}; f_{SEL}; f_{DC}; f_{DL}. We kept all the default parameters for both algorithms (EA and SA), with one exception. As a default the mutation and the neighborhood operators make changes uniformly at random and so they do not take into consideration the current value of the changed vector component of a test candidate. Because f_{AL} and f_{SE} are highly dependent on branch distance and expect a change close to the current value, to make the comparison more fair we defined new mutation and neighborhood operators centered on the current value v: they select a new value uniformly at random in the range $[v - \varepsilon, v + \varepsilon]$, where ε represents 1% of the domain $[-10^4, 10^4]$.

The study was performed on a well-studied benchmark, the triangle program (McMinn, 2004), and on 440 synthetic test targets that were randomly generated. A synthetic test target is a simple program that contains a set of nested branches to be satisfied. To get realistic test targets, every test target is generated carefully, branch by branch. Every branch

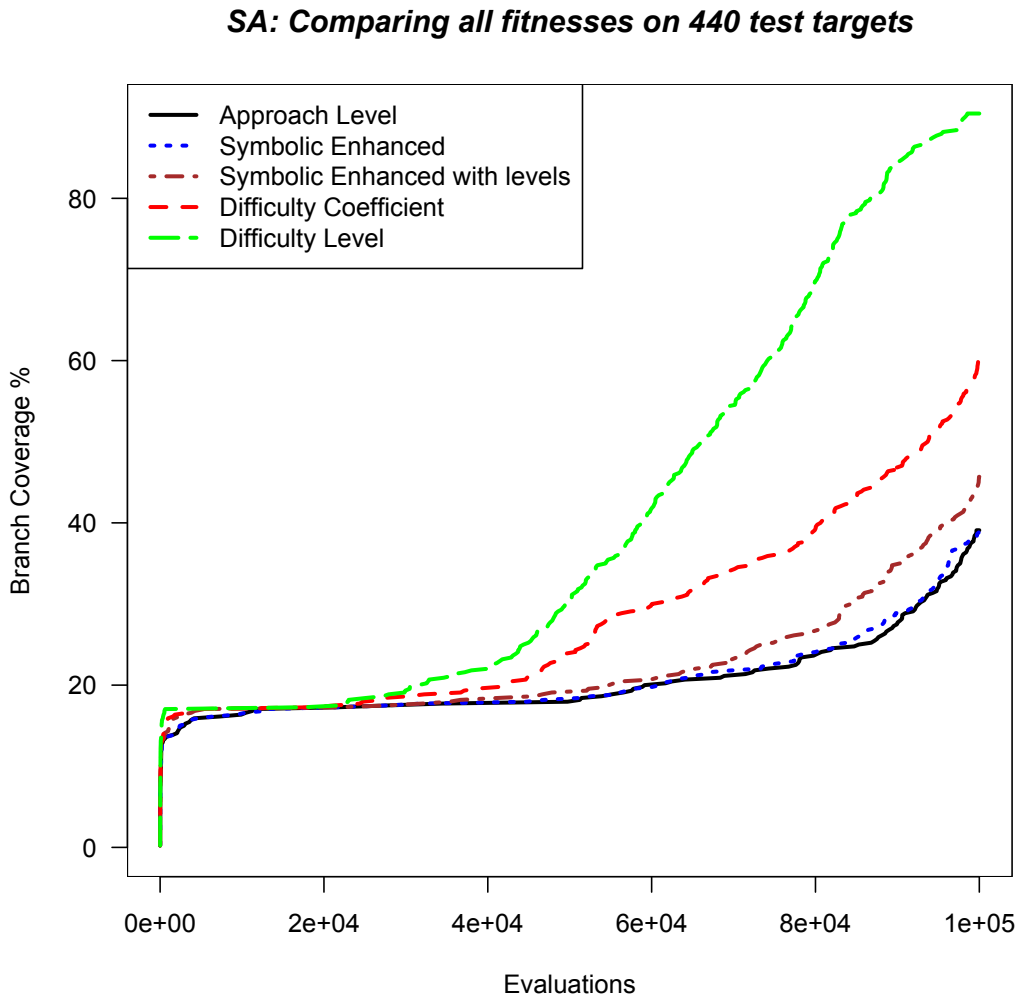

Figure 6.2: SA on 440 test targets

must: i) keep the test target feasible; ii) involve two variables (80%) or a variable and a constant (20%); iii) not be implied by the current test target. [1]

Each search for test data that meets a test target was performed 20 times for every combination of fitness function and meta-heuristic algorithm. If test data was not found after 25000 (respectively 100000) fitness evaluations for EA (respectively SA), the search was terminated. For all techniques, the 20 executions were performed using an identical set of 20 initial populations.

Experiments on the Triangle program show that all fitness functions perform in the same

[1]Benchmarks are available at `http://www.crt.umontreal.ca/~quosseca/fichiers/23-benchsCPAOR13.zip`

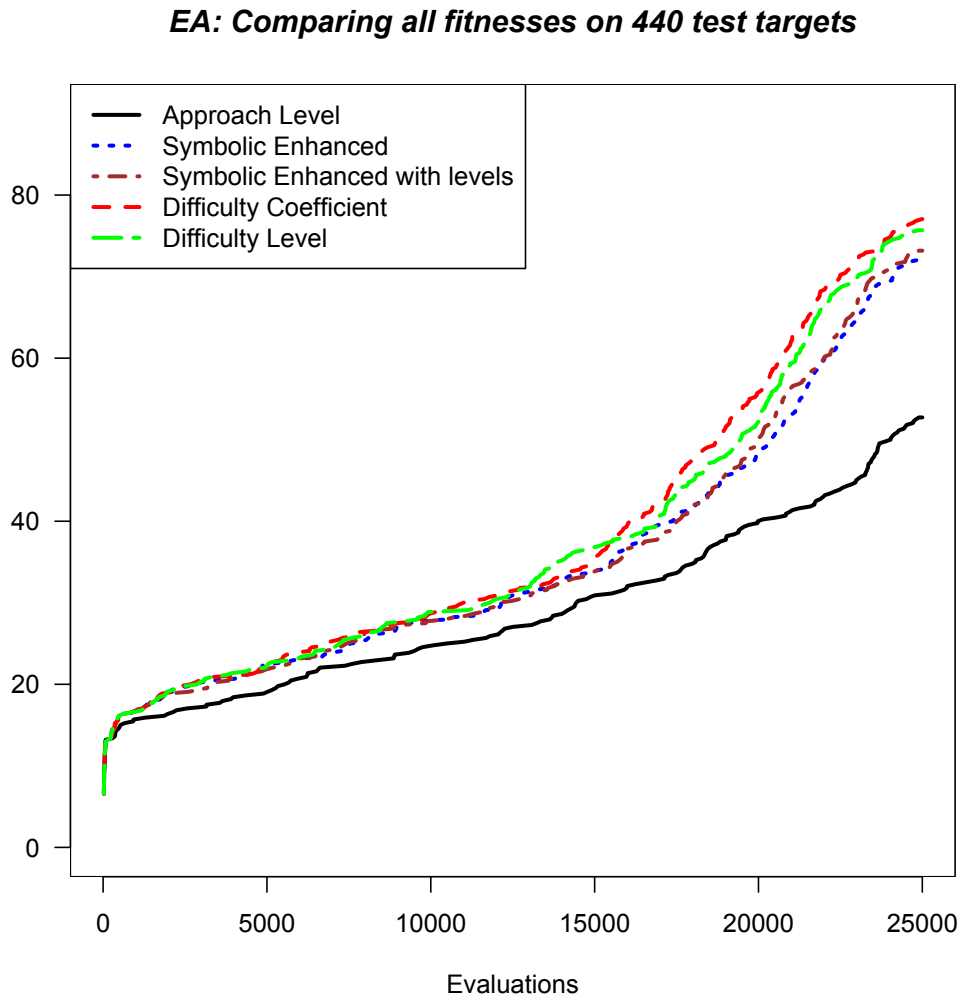

Figure 6.3: EA on 440 test targets

way on all branches except on the two branches that represent an isosceles and an equilateral triangle —they need at least two out of three parameters to be equal. On these two branches we observed that our fitness function f_{DL} outperforms the others in terms of number of evaluations necessary. Therefore f_{DL} makes SB-STDG more efficient on Triangle, especially on these two branches.

Fig. 6.2 and 6.3 show the coverage achieved by the EA and the SA on the 440 test targets with respect to the number of evaluations. For each fitness function, a test target is considered as covered if at least one execution out of 20 succeeded to generate a test data. If more than one execution succeeded to do that, then the median value of evaluations is used. These two plots confirm that the proposed fitness functions can enhance the effectiveness of

SB-STDG: the search based on f_{DL} or f_{DC} is able to cover test targets that are not covered by either f_{SE}, f_{AL} or their combination f_{SEL}. All fitness functions perform similarly on easy test targets but f_{DL} and f_{DC} outperform the rest on hard to reach test targets. The improved effectiveness is particularly clear on the SA results, where we observe that our fitness f_{DL} can reach 90% coverage whereas the best of f_{SE}, f_{AL} and f_{SEL} could not reach 45%.

In both Figures 6.2 and 6.3 we can observe that, from a certain point on, a given number of fitness evaluations allows our fitness functions to cover more targets than f_{SE} or f_{AL}. Therefore we may say that the proposed fitness functions improve the efficiency of SB-STDG.

6.4 Summary

In this chapter we defined two new metrics to measure the difficulty to satisfy a constraint (i.e., branch) in the context of test-data generation for software testing. We used them to propose new fitness functions that evaluate the distance from a test candidate to a given test target. We presented results of an empirical study on a large number of benchmarks, indicating that the search algorithms evolutionary algorithm and simulated annealing with our new fitness functions are significantly more effective and efficient than with the largely used fitness functions from the literature.

Part III

Lightweight Static Analyses to improve SB-STDG

CHAPTER 7

SCHEMA THEORY OF GENETIC ALGORITHMS TO ANALYZE AND IMPROVE SB-STDG

GA have been effective in solving many search and optimization problems. Software testing is one field wherein GA have received much attention to solve the problem of test-data generation. However, few works analyze this problem to see if it is a class of problem where GA may work well. The schema theory is an analysis tool that theoretically explains how and in witch class of problem GAs work well. In this chapter we use the schema theory to analyze the problem of test-data generation. Based on this analysis we propose EGAF-STDG an enhanced GA framework for software test-data generation. It uses a schema analysis to identify performance schemata. To preserve these schemata during evolution phase and maintain their diversity our framework provides a new selection strategy called pairwise-selection as well as adaptive crossover and mutation operators.

We empirically evaluate EGAF-STDG on a set of synthetic programs to compare the branch coverage and number of evaluations required to generate test data with the state-of-the-art of evolutionary testing, i.e., DaimlerChrysler system for evolutionary testing Baresel *et al.* (2002); Wegener *et al.* (2001). We show some preliminary experiments in which a high branch coverage was obtained and the number of fitness evaluations was significantly reduced.

7.1 Schema Theory for Test-data Generation Problem

Test-data generation problem aims to generate an input vector (i.e, individual) that satisfies a set of branch-conditions leading to a test target. Its search space can be split into subspaces where each subspace represents the set of individuals satisfying a given subset of branch-conditions. Thus, individuals are distinguishable by the subset of branch-conditions that they satisfy. Therefore a schema can be defined as a set of individuals that satisfy a meta-condition (i.e., a conjunction of branch-conditions), and its order is equal to the number of conjuncts. An individual i is an instantiation of a schema, if and only if i satisfies its meta-constraint. Thus, the set of schemata that present a problem of test-data generation is all possible branch-conditions whereon the test target is control dependent. For example, an UUT that takes three input arguments x_1, x_2, and x_3, a test target t control dependent on three branches b_1, b_2, and b_3, can be modeled by a subset from the set of schemata given in Table 7.1. Each schema represents an exponential number of sub-schemata, where a sub-

Table 7.1: A set of possible schemata for a test target that is control dependent on three branches (b_1, b_2, b_3).

Schema	Meta-constraint	Pattern	Order
s_{11}	$\{(x_1, x_2, x_3) \mid b_1$ is satisfied$\}$	$b_1 \quad * \quad *$	1
s_{12}	$\{(x_1, x_2, x_3) \mid b_2$ is satisfied$\}$	$* \quad b_2 \quad *$	1
s_{13}	$\{(x_1, x_2, x_3) \mid b_3$ is satisfied$\}$	$* \quad * \quad b_3$	1
s_{21}	$\{(x_1, x_2, x_3) \mid b_1 \wedge b_2$ is satisfied$\}$	$b_1 \quad b_2 \quad *$	2
s_{22}	$\{(x_1, x_2, x_3) \mid b_1 \wedge b_3$ is satisfied$\}$	$b_1 \quad * \quad b_3$	2
s_{23}	$\{(x_1, x_2, x_3) \mid b_2 \wedge b_3$ is satisfied$\}$	$* \quad b_2 \quad b_3$	2
s_{31}	$\{(x_1, x_2, x_3) \mid b_1 \wedge b_2 \wedge c_3$ is satisfied$\}$	$b_1 \quad b_2 \quad b_3$	3

schema is defined by the meta-constraint of the main schema while fixing one input argument to a value from its domain.

Clearly the schema as defined is hierarchical structured: intermediate-order schemata (e.g., s_{21},s_{23}) play the role of stepping stones to go from lower-order schemata (e.g., s_{11},s_{12}) to higher-order schemata (e.g., s_{31}) (Mitchell *et al.*, 1992). This is one essential feature for the *building-blocks hypothesis*, i.e., partial schemata can construct better and better higher-order schemata. Thus, individuals from a schema having a fitness value higher than the average are likely to produce fitter individuals (Holland, 1975). Therefore a fitness function based on the defined schema may work well under certain circumstances. Based on this schema we can define different fitness functions for evolutionary testing. Mitchell *et al.* (1992) termed this class of fitness functions *Royal Road functions*.

7.2 Royal Road Function

To define a royal road function for evolutionary testing we use the concept proposed by Jones (1994). The fitness value is composed of two scores called PART and BONUS. PART considers each building block individually in such a way that each building block receives a fitness value. BONUS is to reward optimal building blocks that reach their optimal fitness value. Because we define a building block by a branch-condition, we can define PART by a straightforward assignment of its normalized branch distance (η) Arcuri (2010). If for some reason an exact evaluation of a building block is impossible (e.g., a non-executed branch and its expression is impossible to evaluate symbolically), then its PART can be overestimated by assigning it the value 1. In this case, if we ignore the BONUS (e.g., BONUS=0) then we fall on the Approach Level fitness function (f_{AL}) (Wegener *et al.*, 2001) and Symbolically Enhanced fitness function (f_{SE}) (Baars *et al.*, 2011) expressions. Note that the main difference between

these two fitness functions is that the latter complements the former by evaluating some non-executed branches. This class of fitness function considers that all branches are directly comparable or balanced, however the branch distances extracted form different branches are more complicated to compare because each branch has its own features that should be considered (e.g., a given branch may be more complex to satisfy than another). In previous work Sakti *et al.* (2013) studied branch-hardness (i.e., the difficulty to satisfy a branch) and showed that the Difficulty Coefficient (DC) is a good corrective metric to get comparable branch distances among different branches. In this chapter, we use this metric to adjust branch distances. Using this adjustment we fall on the Difficulty Coefficient fitness function (f_{DC}) expression (Sakti *et al.*, 2013).

BONUS allows the revised class of royal road functions to outperform the simplest versions of the royal road functions and other search algorithms. Thus, it is key to performance and should not be ignored. Jones (1994) defines the BONUS as a way to classify individuals into distinct levels (i.e., order of schema). The BONUS must be a distinctive value, i.e., from the overall fitness value we can make the distinction between individuals from different order of schemata. This means that the BONUS must be greater than the maximum value that an individual can have from the sum of all PARTs. Since we define PART by a normalized distance that is always upper-bounded by 1, the BONUS can be defined by the number of branches. But this is an equitable reward regardless of the difficulty of satisfying a branch. To take into account the branch-hardness, we use the Difficulty Level (DL) metric (Sakti *et al.*, 2013) to reward an optimal building block.

In general an evolutionary testing fitness function is a minimization function, so the BONUS of each block must decrease its objective. By default the objective value must contain the maximum summed BONUS that may appear in an objective value and when a building block reaches its optimal value the BONUS is decreased. Since the optimal fitness value of a block building is equal to 0, a simple equivalent expression may be obtained by penalizing non-optimal building blocks instead to reward optimal building blocks. Therefore unsatisfied branch is penalized by its BONUS, so a minimization royal road function for evolutionary testing can be expressed as follows:

$$f_{RR}(i, C) = \sum_{c \in C'} \Big(BONUS(c) + PART(i, c) \Big) = \sum_{c \in C'} \Big(DL(c, C) + \eta\big(\delta \cdot DC(c)\big) \Big)$$

- C is the set of branches whereon the test target is control dependent;
- C' is the subset of C that contains unsatisfied branches by i;
- $DL(c, C)$ and $DC(c)$ as are defined in (Sakti *et al.*, 2013).

In evolutionary testing, the level of dependency between building blocks may be very different

from one instance to another. An instance may contain a set of strongly dependent building blocks, however another may contain a set of building blocks that are weakly dependent or totally independent. Because a hierarchical structure must take into account the conflicts among fit schemata, the inter-dependency between building blocks, and intra-dependency within building blocks (Watson *et al.*, 1998), we cannot enforce a general subset of schemata (i.e., a plan to follow) for all instances of the problem of test-data generation.

Note that the proposed formulation of the fitness function does not predict an exact subset of schemata (plan). This is an advantage because it can deal with any instance of the problem of test-data generation, but at the same time this is an inconvenience while its landscape may significantly change from one instance to another. Therefore as any general fitness function the proposed one may be deceptive in some instances of evolutionary testing. The fitness function alone cannot perform well on every instance of evolutionary testing. For further increases in power, in addition to a good fitness function the GA must exploit features associated with good performance (Holland, 1975). In Holland's words *"unexploited possibilities may contain the key to optimal performance, dooming the system to fruitless search until they are implemented. There is only one insurance against these contingencies. The adaptive system must, as an integral part of its search of* "(schemata)"*, persistently test and incorporate structures properties associated with better performance."*

In the following subsections we propose some adaptive GA operators that use schema analysis to detect and exploit some features related to evolutionary testing performance.

7.3 Pairwise Selection

The hope of a GA is finding the fit desired schemata at each generation. A distributed population on schemata may answer this hope. Since fitter individuals may be from a same schema, the selection operator must balance between fitness, diversity, and schemata. In the literature there are several selection operators that balance between fitness and diversity (Goldberg et Deb, 1991). *Pairwise-Selection* is a selection strategy that we propose to boost the selection by taking into consideration the diversity in terms of schemata. To reach this aim the selection operator must be able to identify schemata or distinguish between them. From the overall fitness value it may be possible to know the order of schemata, however it is impossible to identify an exact schema since many schemata may have the same order.

In evolutionary testing, we know a priori the optimal value of a fitness function (generally branch distance is equal to 0), whence the schemata identification is possible by storing the individual fitness value of each building block (branch distance of each individual branch) and using them during the GA evolution phase. A schema detector can be defined as a binary

vector derived from the schema pattern by replacing the asterisks by 0 and constraints by 1. The scalar product of a branch distance vector and detector vector is equal to 0 if and only if the individual is an instance of the schema. Further, the branch distance vectors may be a good mean to measure the similarity or the distance between individuals in terms of schemata. The *similarity* between two individuals is equal to the scalar product of their branch distance vectors. The similarity is maximal if the two individuals are instances of the same schema, it is minimal if the two individuals complement each other (it is equal to 0).

The pairwise selection combines two selection strategies to select individuals in a pairwise manner. The first individual is selected according to a selection strategy that takes in consideration the fitness and the population diversity. The second individual is selected using a selection strategy that is focused on the diversity in terms of schemata. The first strategy sorts individuals according to a linear ranking method (Whitley, 1989; Pohlheim, 2006), after which the first individual is selected using stochastic universal sampling (Baker, 1987; Pohlheim, 2006). The second individual is selected to increase the likelihood of getting a complement of the first individual using a tournament selection (Goldberg et Deb, 1991) based on lowest similarity, and worst fitness in case of ties.

Pairwise selection has a good likelihood of diversifying the population in terms of schemata. Further it couples two individuals that have a potential to generate fitter offspring.

7.4 Schema Analysis

It is well known that crossover and mutation operators disrupt schemata in a population. The schema theorem shows that crossover and mutation rates are the main source of the loss of schemata (Holland, 1975). Those rates are constant in evolutionary testing and are randomly applied without any consideration of schemata. Adaptive rates may help to reduce losses and make evolutionary testing more effective and efficient.

Since crossover and mutation operators perform on genes, comparison in terms of genes' performances is a natural way to define adaptive crossover and mutation operator. The contribution of each gene to the fitness may be an adequate indicator to adapt those two operators. The question is how to determine the contribution of each input variable? To answer this question we propose a schema analysis to identify the influence of each input variable on each schema.

We say that an input variable v, influences a schema s if v influences any basic building-block component in s. Since a basic building block is defined by an individual branch, we keep Korel's definition of influence (Korel, 1990) and we extend it with *indirect influence*. A variable v indirectly influences a branch b if b is influenced by a variable w while a definition of

w is in an execution block B and the execution of B depends on v value. Fig. 9.1 is a sample source code wherein the input variable i indirectly influences the conditional statement at Line 5.

The schema analysis generates an *influence matrix* that summarizes input variables influence on basic building blocks. An *influence matrix* is a UUT's matrix that reflects the dependency relationship between conditional statements and input variables: its lines are input variables and its columns are conditional statements. A cell $[v, c]$ of an influence matrix is equal to 1 if the conditional statement c depends in any way on input variable v, else it is equal to 0. Fig. 7.2 presents the influence matrix for foo function.

The schema analysis is a sort of data flow analysis. Therefore, with a slight modification to a data flow analysis algorithm we can generate the influence matrix. In this chapter, we assume that the control flow graph does not contain cycles (loops are unfolded $kpath$ times) and use an adapted version of the Basic Reach Algorithm (Allen et Cocke, 1976) to generate the influence matrix.

Using the influence matrix an estimated input variable contribution can be determined in each of the branch fitness. We assume that input variables involved in a branch contribute equitably to its fitness value. Therefore a fitness value can be assigned to each input variable by summing its contributions among all branches wherein it is involved in an influence relationship.

7.4.1 Adaptive Crossover Operator.

The aim of any crossover operator is generating at least one offspring which is fitter than each parent. To achieve this objective a crossover operator needs to detect the origin of the fitness in each parent and exploit them to get fitter offspring. An adaptive uniform operator can be developed for this propose that focuses only on one offspring wherein the fitter genes are recombined. The adaptive uniform operator generates a main offspring and a secondary offspring. The main offspring receives from the first parent all genes that have reached the optimal fitness value then the remaining genes are chosen from both parents according to fitness values of genes (i.e., the fitter gene of both genes is chosen). The secondary offspring receives each gene from one parent with an equal probability.

Despite genes being linked by building blocks and the latter not being necessary independent, this adaptive uniform crossover has a good likelihood of making a fitter offspring and reducing the probability of losing schemata.

```
1  foo(int i,int j,int k,int l ) {
2      int x=0;
3      if(i>0)      //b1
4          x=j;
5      if (x>0 && k>0)  //b2
6          if(l>0)    //b3
7              return 1;
8
9    return 0;
10 }
```

Figure 7.1: foo function

	b_1	b_2	b_3
i	1	1	0
j	0	1	0
k	0	1	0
l	0	0	1

Figure 7.2: influence matrix for foo function

7.4.2 Adaptive Mutation Operator.

Two features define a mutation operator: the probability of changing a gene and the way a gene is changed. A mutation operation can be considered successful if the new offspring is fitter than its parent. To achieve this a mutation operator needs to detect the sources of weakness from the fitness of the parent and change them in an adequate way. An adaptive mutation operator can be developed to answer this. Instead of using an equal probability $1/\ell$ to change each gene, the adaptive mutation operator uses a probability computed in terms of the fitness value scored by each gene. It makes a distinction between two groups of genes: optimal genes that have the optimal fitness value (0) and non-optimal genes that do not yet reach the optimal fitness value. Only genes from the second group are subject to mutation with an equal probability $\frac{1}{|Sg|}$, where $|Sg|$ is the number of genes in the second group (non-optimal). Thus a mutated value is computed in terms of the *min* and the *max* of fitness values of branches (building block) wherein the input variable (gene) is involved. A value r is randomly selected from one of three intervals $\{[1, min], [min + 1, max], [max + 1, uBound]\}$ with probabilities $\{0.6, 0.3, 0.1\}$, then this value is added or subtracted to the current gene value. A minimum width is required for each interval (e.g., for integers each interval must contain at least 50 values otherwise the interval is modified to contain this number starting from its lower bound).

7.5 Empirical Study

The main goal of our empirical study is to compare our proposed enhanced GA framework against the state of the art and to analyze its impact in terms of effectiveness and efficiency on evolutionary testing. Evolutionary testing is considered more *efficient* if the enhanced GA framework is able to reduce the number of evaluations to achieve a given coverage; it is considered more *effective* if the enhanced GA framework is able to cover more targets. To

get a more revealing analysis we evaluate separately the four components in our enhanced GA framework: the royal road function of evolutionary testing, the pairwise selection, the adaptive crossover operator, and the adaptive mutation operator. This allows us to determine the contribution of each component in the enhanced GA framework and to analyze each research hypothesis. More precisely this case study aims at answering the following four research questions:

(RQ1): Validation of evolutionary testing royal road function. Can an evolutionary fitness function expressed according to the schema theory make evolutionary testing more efficient or effective? This shows the impact of a fitness function expressed according to the schema theory;

(RQ2): Validation of schemata diversity hypothesis. Is the pairwise selection able to make evolutionary testing more efficient or effective? This shows the impact of maintaining the population diversity in terms of schemata;

(RQ3): Performance of proposed adaptive crossover. Can the adaptive crossover operator that combines fitter genes enhance evolutionary testing in terms of efficiency of effectiveness? This shows the effect of preserving schemata by exploiting the sources of performance;

(RQ4): Performance of proposed adaptive mutation. Can the adaptive mutation operator that assigns high probability to "weaker" genes enhance evolutionary testing in terms of efficiency of effectiveness? This shows the impact of incorporating structures and properties associated with weaker performance to preserve schemata and mutate individuals.

To ensure that we compare, indeed, with the state of the art a careful implementation of the DaimlerChrysler system for evolutionary testing (DCS-ET) (Baresel *et al.*, 2002; Wegener *et al.*, 2001) was set up by using the evolutionary computation for Java (ECJ) system (Luke *et al.*, 2010). We choose ECJ because it is founded on the same principles as DCS-ET: both systems support breeding based on multi-populations, or Breeder Genetic Algorithm (Mühlenbein et Schlierkamp-Voosen, 1993). The ECJ features have made our task reimplementing DCS-ET easy to achieve. DCS-ET has been largely studied and reimplemented in the literature and it is always considered as the state of the art (Harman et McMinn, 2010). In the next paragraph we give a brief description of DCS-ET (Wegener *et al.*, 2001; Harman et McMinn, 2010).

The population contains 300 individuals that are uniformly distributed over six sub-populations. All sub-populations are evolved in parallel separately according to the following

process: (1) Evaluation: according to a fitness function, a score is assigned to each individual; (2) Selection: individuals are selected using a linear ranking (Whitley, 1989; Pohlheim, 2006) with a selection pressure equal to 1.7 followed by stochastic universal selection (Baker, 1987; Pohlheim, 2006); (3) Crossover: parents are recombined by using a discrete recombination (Mühlenbein et Schlierkamp-Voosen, 1993); (4) Mutation: offsprings are mutated by the breeder genetic algorithm mutation strategy (Mühlenbein et Schlierkamp-Voosen, 1993); (5) Reinsertion: an elitist reinsertion strategy is adopted — it keeps the top 10% of the current population; (6) Random Exchange: every 20 generations sub-populations randomly exchange 10% of their individuals with one another; (7) Populations competition: every four generations, the populations are ranked according to their progress in terms of performance (mean fitness), and the size of each sub-population is updated, with weaker sub-populations losing individuals in favor of stronger ones.

We enhanced DCS-ET by f_{RR}, pairwise selection, adaptive crossover, and adaptive mutation operators to get a first version of our framework EGAF-STDG.

The study was performed on 440 synthetic test targets that were randomly generated. A synthetic test target is a simple Java program that contains a set of nested branches to be satisfied (i.e., only the deeper branch is targeted). The search space and the number of nested branches are the main factors in determining the difficulty of an evolutionary testing problem. The search space grows exponentially with the number of input variables. To get simple Java programs that represent different levels of difficulty the number of input variables is varied from 2 to 9 and each test target contains at least a number of nested branches equal to the number of input variables and at most is equal to double the number of input variables. For each combination of the number of input variables and number of nested branches 10 simple Java programs are generated. We define each input variable as an integer that takes its value from the large domain $[2^{-30}, 2^{30}]$. Each simple Java program denotes a separate search problem for which the size of the search space approximately ranges from 2^{62} to 2^{279}. To get realistic test targets, every test target is generated carefully, branch by branch. Every branch must: i) keep the test target feasible; ii) involve two input variables (80%) or an input variable and a constant (20%); iii) not be implied by the current test target.[1]

Each search for test data that meets a test target was performed 20 times. This repetition allows reducing the random aspect in the observed values. A set of 20 random number seeds was used to seed the random number generators. If test data was not found after 30000 fitness evaluations for both DCS-ET and EGAF-STDG, the search was terminated.

[1]Benchmarks are available at `http://www.crt.umontreal.ca/~quosseca/fichiers/23-benchsCPAIOR13.zip`

7.5.1 Analysis

Table 7.2 summarizes the results of 8800 executions for each combination of framework, BONUS, and PART. An execution is considered successful if it could generate a test datum. To compute a maximum coverage (Max. Cov.), a branch is considered covered if test data were found during at least one of the 20 executions. To compute a average coverage (Ave. Cov.), a branch is considered covered if test data were found during at least teen of the 20 executions. The f_{RR} with PART equal to the adjusted branch distance ($\delta \cdot DC$) outperforms f_{RR} with the simple branch distance (δ). Thus in our framework (EGAF-STDG) f_{RR} with BONUS different from zero works better than with BONUS equal to 0, further f_{RR} with DL is better than with $\mid C \mid$. Contrary to our framework on DCS-ET the f_{RR} with DL or $\mid C \mid$ slightly decreases the performance. This is because the f_{RR} with BONUS different from 0 favors higher-order schemata and these latter are overlapped, but DCS-ET uses simple crossover and mutation operators that disrupt schemata. Thus, **we answer RQ1 by claiming that the evolutionary testing royal road function can make evolutionary testing more effective and efficient if the schemata are preserved and their diversity is maintained**.

To measure the advantage of each proposed component the framework DCS-ET is modified step by step. First we replaced DCS-ET's selection strategy by our pairwise selection, then we changed the crossover operator to our adaptive one. After that we modified the mutation rate of DCS-ET by making it adaptive. Finally we replaced the mutation way by our proposed one. Figure 7.3 summarizes different changing steps that we did to pass from DCS-ET to EGAF-STDG.

For our benchmark set of simple programs, in Table 7.2 and Figure 7.3 it is clear that our framework significantly outperforms the DCS-ET framework both for branch coverage and number of fitness evaluations required to cover a same number of branches. Also each proposed operator alone performs better than its simple counterpart. The pairwise selection could not show a significant enhancement in terms of branch coverage, but it is distinctly better than the selection strategy used by DCS-ET in terms of number of evaluations. The pairwise selection offers to the crossover operator a pair of individuals that complement each other. According to the results the complementary between the crossed individuals may diversify schemata and preserve them. Therefore, **we answer RQ2 by claiming that the pairwise selection is able to maintain the population diversity in terms of schemata, thereby making evolutionary testing more efficient but not more effective**.

The proposed adaptive crossover does not show a meaningful difference compared to the discrete crossover because of the complicated relationship between schemata: the latter are

Table 7.2: Results of different expressions of the evolutionary testing royal road function on both frameworks.

f_{RR}		SUCCESS		EVALUATIONS		Max. Cov.		Ave. Cov.	
BONUS	**PART**	**DCS**	**EGAF**	**DCS**	**EGAF**	**DCS**	**EGAF**	**DCS**	**EGAF**
0	$\eta(\delta)$	5,084	8,511	175,619,700	43,269,900	78.18	99.77	58.63	97.27
	$\eta(\delta \cdot DC)$	5,262	8,521	173,914,800	42,747,300	80.45	99.09	60.00	97.72
$\mid C \mid$	$\eta(\delta)$	5,048	8,515	175,827,000	43,021,200	77.72	99.54	56.59	97.27
	$\eta(\delta \cdot DC)$	5,248	8,528	173,948,400	42,644,100	78.86	99.77	58.86	97.50
DL	$\eta(\delta)$	5,037	8,547	176,110,800	41,990,700	76.59	100.00	58.18	97.72
	$\eta(\delta \cdot DC)$	5,166	8,556	175,356,000	41,851,500	79.09	100.00	59.31	97.72

overlapped and strongly depend on each other. The dependency between schemata preserves the combination of fitter genes to generate fitter schemata. This result may be due to the kind of UUTs tried, which essentially use integer data types and their branches are highly dependent on each other. More evidence is needed to draw a general conclusion about the proposed crossover. Yet, on the selected UUTs and the proposed crossover operator **we answer RQ3 by claiming that combining fitter genes does not necessary generate fitter schemata**.

In the adaptive mutation operator, both the adaptive mutation rate (probability) and the adaptive mutation strategy (value) significantly enhance the performance in terms of fitness evaluations and branch coverage. The adaptive mutation rate orients the mutation operator on the "weaker" genes by assigning them a high mutation rate, then allows it to outperform its counterpart that uses equal probability regardless of gene performance. Thus the adaptive mutation strategy orients the mutation operator to choose potential values by using the minimum and the maximum branch distances while favoring the neighborhood. This mutation strategy outperforms the breeder mutation operator in terms of branch coverage and fitness evaluations. Therefore, **we answer RQ4 by claiming that incorporating structures and properties associated with weaker or better performance to preserve schemata and mutate individuals can significantly enhance evolutionary testing**.

7.6 Summary

In the last two decades, search based testing and in particular evolutionary testing have been extensively applied to solve the problem of automated test data generation. However despite the importance of the schema theory analysis for GA, its application to evolutionary testing has posed many challenges. In order to establish a GA framework based on schema theory analysis, this chapter provides a novel automated evolutionary testing framework. We (1)

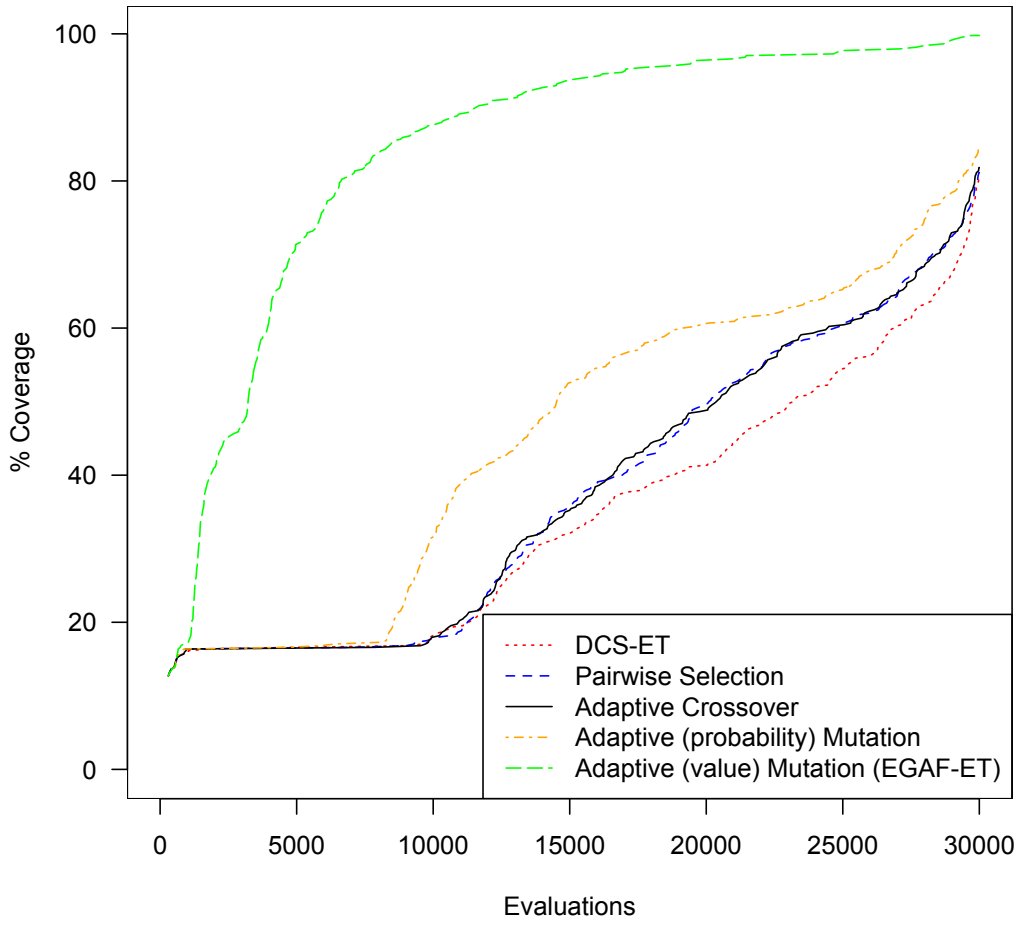

Figure 7.3: The branch coverage in terms of the average number of fitness evaluations.

have adapted schema theory for evolutionary testing, (2) have defined a royal road fitness function for evolutionary testing, (3) have proposed a new selection strategy called pairwise selection that combines two selection strategies to maintain the population diversity in terms of schemata, (4) have proposed an adaptive crossover that dynamically changes its crossover rate according to the schemata, and (5) have proposed an adaptive mutation operator that dynamically changes its mutation rate and the way it mutates in terms of the schemata and their performances. Preliminary experiments have been carried out on some randomly generated benchmark programs. A preliminary case study was carried out to frame the research questions. Results indicate that (a) the proposed royal road function can make evolutionary testing more effective and efficient if the schemata are preserved and their diversity is

maintained; (b) the pairwise selection is able to maintain the population diversity in terms of schemata, thereby making evolutionary testing more efficient but not more effective; (c) the proposed adaptive crossover does not show a meaningful difference comparing to the discrete crossover, thereby combining fitter genes does not necessary generate fitter schemata; (d) incorporating structures and properties associated with weaker or better performance to preserve schemata and mutate individuals can significantly enhance evolutionary testing.

CHAPTER 8

FURTHER STATIC ANALYSES TO IMPROVE SB-STDG

Search-based approaches have been extensively applied to solve the problem of software test-data generation. Yet, test-data generation for object-oriented programming (OOP) is challenging due to the features of OOP, e.g., abstraction, encapsulation, and visibility that prevent direct access to some parts of the source code. To address this problem we present IG-PR-IOOCC, a new automated search-based software test-data generation approach that achieves high code coverage for unit-class testing. We first describe how we structure the test-data generation problem for unit-class testing to generate relevant sequences of method calls. Through a static analysis, we consider only methods or constructors changing the state of the class-under-test or that may reach a test target. Then we introduce a generator of instances of classes that is based on a family of means-of-instantiation including subclasses and external factory methods. It also uses a seeding strategy and a diversification strategy to increase the likelihood to reach a test target. Using a search heuristic to reach all test targets at the same time, we implement our approach in a tool, JTExpert, that we evaluate on more than a hundred Java classes from different open-source libraries. JTExpert gives better results in terms of search time and code coverage than the state of the art, EvoSuite, which uses traditional techniques.

8.1 Instance Generator and Problem Representation

A test-data generation problem is an instantiation of the CUT and a sequence of method calls on that instance. The sequence of method calls can be split into two subsequences: (1) putting the CUT instance in an adequate state; (2) targeting the test target. Because the first subsequence aims to change the object state, we call it *state-modifier* methods and we call the second subsequence *target-viewfinder* methods because it aims to reach a test target. Thus, a representation of the test-data generation problem can be split into three main components: a *CUT-Instantiator*, a sequence of *state-modifier* methods, and a *target-viewfinder* method.

8.1.1 Instance Generator

In the object-oriented paradigm, generally, calling a constructor or a method requires some instances of classes. Given the large number of constructors and method calls needed for solving a testing problem, the generation of instances of classes requires a particular attention.

It is key to successful test-data generation because without an adequate instance of a CUT or a needed objects the solving process may fail before it starts.

Means-of-instantiation

In our approach, we use the term *means-of-instantiation* to represent any means that allows generating an instance of the CUT or more generally to instantiate a class. A means-of-instantiation can be a constructor, a factory method, an accessible data member that is an instance of the required class, or a method defined in another class that returns an instance of the required class.

Means-of-instantiations can be categorized in two groups: internal and external. For a given class c, a means-of-instantiation is internal if it is offered by c itself, i.e., defined in c. A means-of-instantiation is external if it is defined in a class different from c.

To generate an instance of a given class our Instance Generator takes into account all accessible means-of-instantiation offered in a program and according to our diversification strategy one of them is selected. Thus, to instantiate a given class, our instance generator considers five different families of means-of-instantiation: (1) All accessible constructors (if there is no constructor defined the default one is considered); (2) all factory methods, i.e., all statics methods member returning an instance of that class; (3) all statics fields that are instances of the class; (4) all external methods that return an instance of that class, i.e., methods that return an instance of a needed class and are defined outside of that class; (5) recursively all means-of-instantiations of subclasses.

External Factory Methods and Anonymous Class Instantiation: In general, to instantiate a given class, only the internal means-of-instantiations are considered (i.e., constructors, factory methods, and fields). However, external factory methods, i.e., a method that returns an instance of a class and is defined outside of that class, may be a potential means not only for generating instances but also for diversifying the generated instances. Also, in some cases it may be the only solution to instantiate a class. For example, in Figure 8.1, an anonymous class defined at Line 4 is nested in the method `keyIterator`: there is a very weak likelihood to cover branches in that anonymous class without instantiating it as a CUT because all of its methods require an explicit call. One possible mean-of-instantiation of that anonymous class is the method `keyIterator`, which returns an instance of that class. Once that anonymous class is instantiated reaching its branches becomes a simple search problem.

An anonymous class is instantiable if and only if the method wherein it is declared returns its instance. Using such mean-of-instantiation, we can test an anonymous class separately and directly call all of its public methods.

```
1  public Iterator<K> keyIterator() {
2    reap();
3    final Iterator<IdentityWeakReference> iterator =
4            backingStore.keySet().iterator();
5    return new Iterator<K>() {
6      private Object next = null;
7      private boolean nextIsSet = false;
8      public boolean hasNext() {
9        ...
10     }
11     public K next() {
12       ...
13     }
14     public void remove() {
15       ...
16     }
17     private boolean setNext() {
18       ...
19     }
20   };
21 }
```

Figure 8.1: Skeleton of an anonymous class that can be instantiated, from class `org.apache.lucene.util.WeakIdentityMap`

Subclasses: Subclasses (stubs) are always used to instantiate abstract classes or interfaces but rarely to instantiate a class that offers some other means-of-instantiation (e.g., constructors). However, in some cases, using an instance of a subclass may be the only means to cover protected methods or methods implemented in abstract classes. Also, using means-of-instantiations offered by subclasses may significantly diversify generated instances, especially for arguments.

Diversification Strategy

To diversify the generated instances of a given class, we assume that its means-of-instantiations can split the search space into some subspaces where each means-of-instantiation is represented by a subspace. Generating required instances from different subspaces may increase the diversity and the likelihood to reach a test target. To enlarge the number of subspaces and have more diversity our Instance Generator takes into account all accessible means-of-instantiations offered in a program.

In the presence of different means-of-instantiation, choosing between them to generate a certain number of instances is problematic. For example, in the Joda-Time[1] library, most classes have more than a hundred means-of-instantiations, e.g., the class `org.joda.time.base.`

[1]Joda-Time: An open source library that provides a quality replacement for the Java date and time classes. Available at http://www.joda.org/joda-time/

`AbstractPartial` and the interface `org.joda.time.ReadablePartial` each have 225 different means-of-instantiations through their subclasses. The complexity of these means-of-instantiations vary greatly. For example, a mean-of-instantiation that does not need parameters is less complex than another that needs some instances of other classes, that needs an instance of a class that is difficult to instantiate, or, for accessibility reason, that is "not instantiable". In some cases, instantiating a class using one of its complex means-of-instantiations may be harmful, i.e., may negatively influence performance of a test-data generation search (e.g., a means-of-instantiation involves complex computation operations in its implementation and requires significant time to be executed).

To balance diversity and complexity our selection strategy favors less complex means-of-instantiations while diversifying generated instances. A probabilistic selection strategy is implemented for this propose that diversifies the generated instances without compromising performances. Such a selection strategy needs an evaluation of the complexity of a means-of-instantiation.

A given means-of-instantiation mi can be considered complex if it fails to generate an instance of a class because of one of the following reasons:

- mi involves complex computation;
- a precondition on its arguments is hard to satisfy;
- recursively one of its needed arguments is complex to instantiate.

To simplify the measurement of the complexity of a means-of-instantiation, we divide it into two complexities: the complexity to execute a means-of-instantiation and the complexity of instantiating its needed arguments. Initially, we suppose that the complexity to execute a means-of-instantiation or instantiating a class is constant and equal to a constant IC (Initial Complexity, equal to 20 in our approach). Formally an initial complexity of a means-of-instantiation is measured according to the following formula:

$$C^0_{mi} = (Nbr\_Arg + 1) \times IC$$

This expression uses the number of arguments as a measure to evaluate the complexity of a means-of-instantiation. Sometimes, preconditions or the complexity of instantiating an argument may make a mean-of-instantiation that needs only one argument more complex than another that requires many arguments. To take this observation into consideration, we use the percentage of failure of generating instances to measure the complexity of a means-of-instantiation, i.e., we attempt to generate a number of instances using the same means-of-instantiation while observing the number of failures. Using such a computation, we obtain an

accurate measure. However, evaluating the complexity of all means-of-instantiations before the search may be expensive. To simplify the computation, we measure the complexity of a means-of-instantiation on the fly while searching test data: initially, a complexity of a given means-of-instantiation mi is evaluated to C^0_{mi}, each time mi is called to generate an instance of its returned class, its complexity measure is updated based on failures.

Our updating of the complexity measures is based on a penalty system. We use two types of penalties:

- NON INSTANTIABLE PENALTY (NIP): this penalty is assigned to a class c if our instance generator could not generate an instance of c because c does not have any mean-of-instantiation. This penalty should be big enough to reduce the likelihood (to zero if possible) of the selection of a mean-of-instantiation that needs a non instantiable class. In this work, we use $NIP = 10^6$.

- FAILURE PENALTY (FP): this penalty describes the difficulty of executing a mean-of-instantiation. Every time a mean-of-instantiation could not generate an instance of its returned class, a failure penalty is added to its complexity measure. This may happen if at least one parameter does not satisfy the means-of-instantiation preconditions or time out of instantiation is reached. To allow other means to be selected, this penalty should be bigger than the most complex mean-of-instantiation. In this work, $FP = 10 \times IC$.

Finally, the complexity of a means-of-instantiation is measured, at a time t, as follows:

$$C^t_{mi} = C^0_{mi} + x^t_{mi} \times FP + y^t_{mi} \times NIP$$

where x^t_{mi} represents the number of failures of mi until t; y^t_{mi} represents the number of failures of mi caused by a non-instantiable argument until t.

Then it is possible to define a selection strategy based on the complexity measure to select means-of-instantiations with low complexity. Such a strategy always favors means-of-instantiations with low complexity, which may reduce the diversity of the generated instances. To balance complexity and diversity, we use a cost of diversity (DC_{mi}) that is determined by a penalty system. Each time a means-of-instantiation succeeds to generate an instance of a class, its diversity cost is increased by a constant DIVERSITY PENALTY (DP_{mi}). This constant is defined in term of complexity to keep an advantage for means-of-instantiations with low complexity. Therefore, each means-of-instantiation has its own DP_{mi}, which depends on its complexity. In our approach, DP_{mi} and DC_{mi} are computed as follows:

$$DP^t_{mi} = \begin{cases} 0 & mi \text{ fails} \\ \\ C^t_{mi} - C^0_{mi} + IC & mi \text{ succeeds} \end{cases}$$

and:

$$DC^t_{mi} = \sum_{t' \leq t} DP^{t'}_{mi}$$

To instantiate a given class c, each means-of-instantiation in its set of possible means-of-instantiations Set_MI_c receives an instantiation probability proportionate to its own global cost value $GC^t_{mi} = C^t_{mi} + DC^t_{mi}$ and the total cost value of all other means-of-instantiations $TOTAL\_COST^t = \sum_{mi \in Set\_MI_c} GC^t_{mi}$. The instantiation probability to use a means-of-instantiation mi for instantiating c is determined according to the following formula:

$$p^t_{mi} = \frac{TOTAL\_COST^t - GC^t_{mi}}{(|Set\_MI_c| - 1) \times TOTAL\_COST^t}$$

This probability is used for the selection step through a roulette-wheel selection strategy.

Seeding Strategy of Constants

When a branch condition involves constants, covering such a branch may be challenging. To deal with such a problem, many works (Alshraideh et Bottaci, 2006; Alshahwan et Harman, 2011; Fraser et Arcuri, 2012; McMinn *et al.*, 2012b) propose different seeding strategies, especially for strings and primitive types. We adopt a new seeding strategy, inspired by the works (Alshraideh et Bottaci, 2006; Alshahwan et Harman, 2011; Fraser et Arcuri, 2012; McMinn *et al.*, 2012b) but with some differences: in addition to primitive and string, we seed also object constants, and the seeding probability is defined for each data type and each value in terms of the number of occurrences collected.

Seeding with a Variable Probability: In general, an instance generator seeds extracted constants with a fixed probability, i.e., for a primitive type or a string, a value is randomly selected from the set of extracted constants with a certain fixed probabilities. The study conducted by Fraser et Arcuri (2012) shows that a probability equal to 0.2 gave best results compared to other values. It also shows that in some classes seeding can be harmful. Indeed, if the extracted set of integer constants contains only one value, it is undesirable to see 20% of a population formed of the same value because it substantially reduces diversity. We experimentally observed that using a seeding strategy with a fixed probability equal to 0.2 indeed affects the coverage negatively in some classes, especially if the number of extracted constants is small. Also, it is unbalanced to seed two constants with the same probability,

when one is used a hundred times, whereas the other is used only once in the source code. Thus, each constant must have its own seeding probability according to the number of its occurrences in the source code.

We propose a variable seeding probability that is based on the number of occurrences of constants extracted from the source code. Empirically, we found that a probability equal to 0.05 is better than 0.2 if the number of extracted occurrences of constants is less than 10; otherwise, as in previous works, we use a probability equal to 0.2. Thus, a constant with a large number of occurrences in the source code has a higher likelihood of being selected. For example, if the vector of integer constants exacted from the source code is $\{\langle 5, 2, 4, 5, 5, 1, 2, 3, 4\rangle\}$ then the seeding probability to generate an integer is equal to 0.05 because the number of occurrences is less than ten and the probability to seed the value 5 is equal to $\frac{3}{9} \cdot 0.05$.

Seeding the `null` constant: In general, when constants are discussed, only strings and primitive types are considered, although any object may be a constant and this constant may be extracted from the source code (e.g., array constant is often present in the source code). We consider also the `null` constant. The `null` constant is often involved in a branch's condition that checks for a null object, i.e., it requires an equality between a variable and the `null` constant. This type of condition is difficult to satisfy, but it may become easier with a seeding strategy. In a OOP source code, there is often some branch involving the `null` constant (e.g., $object == null$). If such branches are forgotten, there is a high likelihood to generate null pointer exceptions. For example, in the library Apache Commons-Lang[2], in class `org.apache.commons.lang3.ArrayUtils`, among 1,096 branches 20% (170) have a predicate involving the `null` constant. When we tested this class using EvoSuite (Fraser et Arcuri, 2011, 2013c), only 24 branches out of 170 were covered, i.e., 14% coverage of branches involving the `null` constant. This weak coverage does not mean that EvoSuite does not use the `null` constant at all, but it does not use this constant enough to cover branches involving the `null` constant. We think that EvoSuite does not use a systematic seeding strategy with the `null` constant: perhaps it uses the `null` constant sometimes with some classes or when it meets a difficult class to instantiate. However, to satisfy branches' conditions that involve the `null` constant, it is necessary to seed every class with this constant using an adequate seeding probability. In this work, our instance generator systematically seeds the `null` constant while generating instances of classes with a seeding probability equal to 0.05, i.e., for every one hundred of instances, five null instances are used. We chose a probability equal to 0.05 because we have only one value to seed.

[2]Apache Commons-Lang provides extra methods for classes in the standard Java libraries. Available at http://commons.apache.org/proper/commons-lang/

8.1.2 A Representation of the Test-data Problem

To generate unit-test data using SBST techniques, the main component is the test-data problem representation. The key idea behind our representation of the test-data generation problem is in using a static analysis to determine relevant means-of-instantiations (CUT-Instantiator), state-modifier methods, and target-viewfinder methods, and then use them to generate test-data candidates.

CUT-Instantiator

To reach a test target in a non-static method, an instance of the CUT is required. Different means can be used to generate that instance. If the test target is in a constructor or accessible only via a constructor, then we call this type of constructor *CUT-Instantiator*. For a given test target in a CUT, two reasons may make a constructor a CUT-Instantiator: (1) through it the test target is reachable or (2) through it a data member can be modified.

A means-of-instantiation is considered a CUT-Instantiator if and only if it contains the test target, contains a statement that modifies a data member, or calls an inaccessible method directly or transitively via inaccessible methods and the latter contains the test target or contains a statement that modifies a data member.

We denote the set of all CUT-Instantiator of a test target t in a CUT c by $MI_{c,t}$. If $MI_{c,t}$ is not empty then to generate potential instances of the CUT to reach t, only CUT-Instantiators in $MI_{c,t}$ are considered, otherwise all means-of-instantiations of the CUT are considered.

State-modifier Methods

Because of encapsulation, in general, the state of an instance is not directly accessible. To address this accessibility problem, in a test datum, a sequence of method calls is used to put an instance of a CUT in an adequate state by modifying some data members. Because the aim of the sequence of method calls is changing the state of the CUT, instead of exploring random sequences of method calls as previous works, we focus on methods that may modify a data member.

To change the state of an instance of a CUT, we define *state-modifier* methods as all accessible methods that may directly or indirectly assign to some data members a new value, instantiate them or change their states by invoking one of their methods.

An accessible method is a state-modifier method if and only if it contains a statement that modifies a data member or it calls an inaccessible method directly or transitively via inaccessible methods and the latter contains a statement that modifies a data member.

```
1  public class A{
2      private Map<String,Integer>Dm1;
3      private double Dm2;
4      public A(B b, C c, int i ) {···}
5      public void setDm1(String s, int i ) {Dm1.put(s,i);}
6      public void remDm1(int i) {Dm1.put(s,i);}
7      public void setDm2(double d) {Dm2=d;}
8      public mTV(C c, String s, int i) {
9          ···
10         mUT(c, "mTV");
11         ···
12     }
13     private mUT(C c, String s) {
14         ···
15         //test target
16         ···
17     }
18     ···
19 }
```

Figure 8.2: Example of CUT

For example, in Fig. 8.2, the methods $setDm1$ and $remDm1$ are state-modifier methods because they change the state of data member $Dm1$.

Generally, in a given class, not all methods are state-modifier. Thus, using the subset of state-modifier methods instead of all methods to generate the sequence of method calls may significantly reduce the number of possible sequences, i.e., the search space. We denote the set of all state-modifiers for the i^{th} declared data member in a class c by $SM_{c,i}$.

Target-viewfinder Methods

A test target is either in an accessible or an inaccessible method of a CUT. Thus, it may not be directly accessible. In general, in a test datum, the last method in the sequence of method calls is called in the hope to reach the test target. Because a test target is already known, instead of calling a method randomly, we focus only on methods that may reach the test target. A target-viewfinder method aims to reach the test target.

An accessible method is considered a target-viewfinder method if and only if it contains the test target or it calls an inaccessible method directly or transitively via inaccessible methods and the latter contains the test target.

For example, in Fig. 8.2, if we consider that the test target is reaching Line 15, then the accessible method mTV is a target-viewfinder because it is accessible and calls the inaccessible method mUT that contains the test target. We denote the set of all target-viewfinder methods for a given test target t in a CUT c by $TV_{c,t}$.

Static Analysis

For a CUT we use static analysis on the source code to extract its set of CUT-Instantiators, state-modifier methods for each data member, and target-viewfinder methods. To determine these three sets, static analysis identifies branches that modify the data members and execution paths ascending from a branch to an accessible method. A branch can modify a given data member dm if it contains a constructor call that assigns a new instance to dm; an assignment statement wherein dm is on the left side; or, a method call on dm. Thus, static analysis generates three types of information: (I1) information about all branch-modifiers for each data-member; (I2) information about the parent of each branch; (I3) information about all branch-callers for each inaccessible method. Using I1, I2, and I3, static analysis can generate the set of state-modifier methods and a subset of CUT-Instantiators. When a test target is defined, using I2 and I3, the set of CUT-Instantiators is completed and the set of target-viewfinder methods is generated.

Domain-vector

A domain-vector represents the general skeleton of potential test-data candidates. For a given test target, it is a vector composed of the set of CUT-Instantiators, followed by the set of state-modifier methods for each data member and ending with the set of target-viewfinder methods. Correspondingly, a solution is a vector that assigns to each component one instance from its domain with a fixed list of needed instances of arguments. For a nested class (i.e, local, member or anonymous class), its vector is extended with the parent class's vector. We fix an argument or input data by recursively defining its means-of-instantiations. Except for the target-viewfinder method, any component in a solution vector can take an *empty* assignment. Also, the means-of-instantiation of CUT in a solution vector can be empty if and only if all methods in the CUT are static.

Thus, a presentation of a problem of test-data generation that aims to cover a target t in a class c is as follow:

$$\begin{aligned} DV_{c,t} \quad &\longmapsto \langle \{MI_{c,t} \cup \{empty\}\}, \{SM_{c,1} \cup \{empty\}\}, \\ &..., \{SM_{c,n} \cup \{empty\}\}, \{TV(c,t)\}\rangle \end{aligned}$$

For example, if the class A in Fig. 8.2 is under test and the test target is Line 15 and the constructor defined at Line 4 initializes a data member, then the domain-vector is defined as: $DV_{A,15} = \langle MI_{A,15}, SM_{A,1}, SM_{A,2}, TV_{A,15}\rangle$

with:

- $MI_{A,15} = \{A(B, C, int)\}$

- $SM_{A,1} = \{setDm1(String, int), remDm1(int), empty\}$
- $SM_{A,2} = \{setDim2(decimal), empty\}$
- $TV_{A,15} = \{mTV(C, String, int)\}$

In this example, a test data necessary uses the constructor $A(B, C, int)$ to instantiate the CUT and calls $mTV(C, String, int)$ to reach the test target. To change the state of the generated instance, a test datum calls the methods $setDm1(String, int)$ or $remDm1(int)$, then it calls the method $setDim2(decimal)$.

This example shows that the general form of a test datum is almost fixed by the domain vector; it remain only a search algorithm to find adequate arguments to reach the test target.

8.2 Implementation

We have implemented our approach in a tool, JTExpert, that takes as inputs a classpath and a Java file to be tested. JTExpert automatically produces a set of JUnit test cases Tahchiev *et al.* (2010); JUnit (2013) that aims to cover all branches. JTExpert is completely automated, works on ordinary Java source-code (`.java`), and does not need additional information. An executable of JTExpert with all required libraries is available for download at `https://sites.google.com/site/saktiabdel/JTExpert`.

JTExpert performs two phases: a preprocessing phase and a test-data generation phase.

8.2.1 Preprocessing

The preprocessing phase consists of instrumenting and collecting required information for the second phase. To extract relevant information from the source code, we use static analysis as a first phase before starting test-data generation. The Java file under test is parsed and its Abstract Syntax Tree (AST) is generated. Before extracting data, each node in the AST representing a branch or a method is encoded with its parent branch or method into a unique code, i.e., an integer value.

We implement this phase in two main components: Instrumentor and Analyzers.

Instrumentor

To instrument the Java file under test, its AST is modified to call a specific method on entering each branch. This method call takes as inputs the branch code and notifies the testing process when the branch is executed. After the instrumentation, based on the AST, a new version of the Java file under test is saved and compiled to generate its new Java bytecode file (`.class`).

Analyzers

This component generate the problem representation described in the previous section. To extract the information needed for the problem representation and the instance generator, several explorations of the AST are performed: (1) for each method, we identify the set of all branches callers, i.e., branches wherein the method is called; (2) for each data member, we identify the set of all branches modifiers, i.e., branches wherein the data member is modified; (3) for strings and each primitive type, the set of constant values are saved.

To simplify the implementation of parsing the Java file under test and exploring the AST, we used the parser provided with Eclipse JDT (development tools , JDT). JDT makes our static analysis easy because it allows creating an AST visitor for each required information.

8.2.2 Test Data Generation

The test-data generation phase is the core of JTExpert to find the set of test data that satisfies all-branch coverage criterion. We implement this phase into three main components: Instance Generator, Search Heuristic, and Test Data generator.

Instance Generator

This component implements the generation of means-of-instantiations, the seeding strategy, and the diversity strategy described in the previous section. Algorithm 2 presents the different steps of instance generation and complexity measurement. The algorithm distinguishes between three types of classes: Atomic, Container, and any other type of classes (simple classes). All primitive types, classes that encapsulate only primitive types, and the string class are considered Atomic. Each Atomic class has its own random generator that implements the seeding strategy described in the previous section, which uses the whole domain corresponding to the Atomic class (e.g., a short takes its value from the domain $[-2^{15}, 2^{15} - 1]$). A Container is a standard way for grouping objects (e.g., List and Array), i.e., it is an object that can contain other objects, often referred to as its elements. All containers in Java are either simple arrays, classes derived from the `java.util.Collection` interface, or classes derived from the `java.util.Map` interface. The documentation of the Java Collections Framework[3] gives an exhaustive description of all containers that can be used in a Java program. The current version of JTExpert treats these three types of collections as containers, whereas any other collection that does not derive either from `java.util.Collection` or from `java.util.Map` is considered as a simple class. Containers have many particular features (Visser *et al.*, 2006;

[3]Java Collections Framework Overview: `http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`

Algorithm 2 Instance Generator

Input: c is the class to be instantiated
Output: i_c is an instance of c

```
1: if (c is Atomic) then
2:     i_c ← Atomic(c).getInstance()
3: else
4:     if (c is Container) then
5:         i_c ← Container(c).getInstance()
6:     else
7:         if ( Set_MI_c == ∅ ) then
8:             C_c ← NIP
9:             return ClassNotInstantiable
10:        end if
11:        im ← selectMeans()
12:        for (each p in im.getParameters()) do
13:            i_p ← InstanceGenerator(p)
14:            params.add(i_p)
15:        end for
16:        i_c ← im.getInstance(params)
17:        if ( im.succeeds() ) then
18:            DC^t_mi ← DC^{t-1}_mi + DP^t_mi
19:        else
20:            C^t_mi ← C^{t-1}_mi + FP
21:            if ( attempt < MAX_ATTEMPT) then
22:                i_c ← InstanceGenerator(c)
23:            end if
24:        end if
25:    end if
26: end if
27: return i_c
```

Arcuri et Yao, 2008). Test data that involves a container should be generated in a different way than other objects. Instead of using means-of-instantiaitons to generate a container, we implement a random instance generator to generate some types of container: it randomly selects a bounded length then it recursively calls Algorithm 2 to generate all needed elements. For any other class, at Line 11, the set of all its possible means-of-instantiations is generated and a mean-of-instantiation is selected. To generate this set, for a given class, we use the Java Reflection API (Reflection, 2013) to get the means-of-instantiations offered by that class and the open-source library Reflections (Reflections, 2013) to get subclasses and external factory methods. After getting the set of all possible means-of-instantiations, an instantiation probability is assigned to each means-of-instantiation according to its complex-

ity and a roulette-wheel selection strategy is used to select a means-of-instantiation. Then, all needed arguments are recursively instantiated at Line 13. Finally, at Line 16, an attempt to generate an instance of the class is made, after which the complexity of the selected mean-of-instantiation is updated.

Search Heuristic

We combine the proposed approach with a random search. Although random search is the simplest among the search heuristics, it is largely used in software testing because it may reach a high level of coverage (Arcuri et Yao, 2008; Parasoft, 2013; Andrews *et al.*, 2006; Csallner et Smaragdakis, 2004; Pacheco et Ernst, 2005; Pacheco *et al.*, 2007; Oriat, 2005). It also makes it easier to evaluate our proposal without having to isolate its effect from that of a more sophisticated search.

The common random search relies on generating a candidate solution vector randomly to execute the instrumented class and to reach the targeted branch. It stops either if the generated candidate solution vector executes the targeted branch or a stop condition is reached. Such an implementation may hang the search in difficult or unreachable branches (Fraser et Arcuri, 2013c).

In contrast, in JTExpert, a random search is implemented to target all uncovered branches at the same time: it does not focus only on one branch, instead it generates a candidate solution uniformly at random for every uncovered branch. This implementation is likely to reach a good branch coverage quickly because it does not waste efforts on unreachable branches and it benefits from the significant number of branches that may be covered fortuitously. Lines 4 to 21 in Algorithm 4 are a pseudo-code of the search algorithm implemented in JTExpert. Line 7 calls the generator of sequences of method calls that is presented in Algorithm 3. The later uses the domain-vector and the instance generator to guide the random search. In Algorithm 3, at Line 1, an instance of the CUT is generated using the instance generator and the set of CUT-Generators. Hence the implemented random heuristic benefits from all the features in our instance generator, e.g., the large number of means-of-instantiations. Lines 2 to 10 generate a sequence of state-modifier methods: at Line 4 a method is randomly selected from the current set of state-modifier methods, then all required instances of classes are generated in the loop for at Line 5. Finally, at Lines 12 to 16 a target-viewfinder method is generated.

The instance generator and the domain-vector are at the core of our search heuristic. The domain-vector guides the search by restricting the possible sequences of method calls. The instance generator guides the search by diversifying instances of classes.

Algorithm 3 Generator of Sequences of Method Calls

Input: DV domain vector; B branch to reach
Output: TDC a test data candidate

```
1: TDC.object ← InstanceGenerator(CUT)
2: ntv ← DV.length − 2
3: for ( int i = 0; i < ntv; i++) do
4:     m ← randomly select a method from SM_{CUT,i}
5:     for (each p in m.getParameters()) do
6:         i_p ← InstanceGenerator(p)
7:         params.add(i_p)
8:     end for
9:     TDC.methods.add(m, params)
10:    new params
11: end for
12: m ← randomly select a method from TV_{CUT,B}
13: for (each p in m.getParameters()) do
14:    i_p ← InstanceGenerator(p)
15:    params.add(i_p)
16: end for
17: TDC.methods.add(m, params)
```

Test Data generator

This component operates and coordinates other components to generate test data. It implements the skeleton of the whole process of test data generation. Algorithm 4 presents the different steps of this component to satisfy the all branch coverage criterion for a file under test. First, at Line 1, a file under test is instrumented and a new instrumented version of the file is generated. The file is analyzed at Line 2 to get all relevant information (e.g., constants) needed for the next steps. For each branch to be covered in the file, selected at Line 5, a domain vector dv that represents the problem is generated at Lines 6 by an analyzer A. Lines 7 and 8 represent a guided random generation of a test-datum candidate to reach the branch b. A test-datum candidate tdc is generated at Line 7 and executed at Line 8 using the Java Reflection API (Reflection, 2013). If this tdc covers either b or some uncovered branches, then it is inserted in the set of test data TD at Line 10 and all its covered branches are removed from the set of test targets T. At Line 13, b is removed from T. If b is not covered yet, then it is inserted in a waiting set of test targets WT at Line 15. When the search has run through all branches (i.e., T becomes empty), then a new round starts by initializing T with WT at Line 18. Finally, at Line 22, JDT is used to translate the set of test-data TD into a Java file that contains test cases in JUnit format.

Algorithm 4 Test Data Generation.

Input: U is the unit under test
Output: TD a set of test data (in JUnit Format)

```
1: U' ← instrument(U)
2: A ← analyse(U')
3: T ← A.getBranches2Cover()
4: while ( T! = ∅ && !maxTime) do
5:     b ← randomly select a branch from T
6:     dv ← A.getDV(b)
7:     tdc ← generateMethodsSequence(dv)
8:     execute(tdc)
9:     if (tdc.isTestData()) then
10:        TD ← TD ∪ tdc
11:        T.remove(tdc.getCoveredBranches())
12:    end if
13:    T.remove(b)
14:    if (b ∉ tdc.getCoveredBranches()) then
15:        WT.add(b)
16:    end if
17:    if ( T == ∅ ) then
18:        T ← WT
19:        WT.clear()
20:    end if
21: end while
22: writeJUnitTestCases(TD)
```

8.3 Empirical Study

This section presents our evaluation of JTExpert for exploring restricted bounded method sequences, instantiating classes by using the proposed diversification and seeding strategies, and generating test data. We investigate the advantages and limitations of JTExpert by comparing it with EvoSuite Fraser et Arcuri (2011, 2013c) that uses a genetic algorithm to generate test data, which starts with an initial population of chromosomes (i.e., a set of sequences of method calls) randomly generated from the set of all accessible methods. When an argument is needed, EvoSuite randomly selects a means-of-instantiation to generate an instance. In this study, we use version 20130905 of EvoSuite. We have performed the experiments on a Oracle Grid Engine comprising 42 similar nodes, each of them equipped with 2-dual core CPU 2.1 GHZ, 5GB of memory, a Fedora core 13 x86_64 as OS, and Java Development Kit 7.

Table 8.1: Experimental subjects.

Libraries	**#Java Files**	**#Classes**	**#Methodes**	**#Branches**	**#Lines**	**#Instructions**
Joda-Time	50	87	1,411	3,052	5,876	25,738
Barbecue	18	18	161	323	1,041	14,558
Commons-lang	5	6	366	1,942	2,134	9,139
Lucene	2	4	58	202	262	1,364
All	**75**	**115**	**1,998**	**5,519**	**9,313**	**50,799**

8.3.1 Experimental Setup

Empirical Study Subjects

The study was performed on 115 of classes from four open-source libraries. All the selected CUTs have been previously used as benchmark to evaluate competing tools that participated to the SBST contest of the Java unit-testing tool version 2013 Vos (2013) which EvoSuite won Fraser et Arcuri (2013b). Some of these classes were also used in evaluating μTest Fraser et Zeller (2012), which has become a part of EvoSuite Fraser et Arcuri (2013c). The set of classes in this benchmark are carefully selected by the SBST contest committee of the Java unit-testing tool to represent different challenges of unit-testing. Most of those classes are complex for test generation Fraser et Zeller (2012). For example, in the library Joda-Time, accessibility is a problem for several classes: only default access is possible, i.e., classes are visible only in their packages. Also, many classes are difficult to access because they are private and embedded as class members in other classes. Others are difficult to instantiate because they do not offer any accessible constructor and can only be instantiated through factory methods or accessible data members Fraser et Zeller (2012).

Table 8.1 lists the Java libraries that we used in our study. Each line presents one of the libraries while columns show the library names, numbers of Java files under test, numbers of classes, numbers of methods, numbers of branches, numbers of lines of code, and numbers of instructions. These metrics are computed at the byte-code level using the JaCoCo tool Hoffmann *et al.* (2012), so that no empty or comment line is included.

Procedure

In the SBST contest 2013 Vos (2013), tools were evaluated using two metrics: code coverage and mutation score. In this study, we evaluate the two tools, JTExpert and EvoSuite, only with code coverage because mutation score does not apply to our approach and the current version of JTExpert does not generate assertions to kill mutants.

For every class in the benchmark, we generate complete test suites to cover all branches

with JTExpert and compare them with those generated by EvoSuite. Each tool applies on a different level of source code: EvoSuite applies on Java bytecode (`.class`), whereas JTExpert applies on Java source code (`.java`), which may generate differences in the observed coverage of each tool because (1) the Java compiler translates some instructions over boolean into conditional statements and (2) one conditional statement in the source code may be translated into many conditional statements if it contains many clauses. To compare the approaches needed, instead of the coverage measured by each tool, we use JaCoCo Hoffmann *et al.* (2012) to measure the coverage: JTExpert and EvoSuite generate test suites and JaCoCo takes them and measures their coverage, at the bytecode level, in terms of four metrics: method coverage, branch coverage, line coverage, and instruction coverage. Each search for test data that meets branch coverage for every CUT is performed 20 times. This repetition allows reducing any random aspect in the observed values. A set of 20 random number seeds was used to seed the random number generators. To make the experimentation scalable, for each execution, a maximum of 200 seconds is allowed per search including the instrumentation and preliminary analysis stages. If this time is spent by JTExpert or EvoSuite, the tool is asked to stop and, after a maximum of 5 minutes, its process is forced to stop (killed). We stop at 200 seconds because we observe empirically that, after this duration, the coverage progress of each tool becomes very slow. In total, this experiment took $(77 \times 20 \times 200 \times 2) = 616 \times 10^3$ seconds, i.e., more than seven days of computational time.

During all experiments, except the time-out parameter, EvoSuite is configured using its default parameter settings, which according to Arcuri and Fraser Arcuri et Fraser (2011) work well. Also, we configure EvoSuite to skip its test-case optimization phase that consists of optimizing the number of generated test cases because this optimization slows down EvoSuite and does not impact coverage.

Comparing JTExpert to EvoSuite

We compare JTExpert and EvoSuite using box-plots that represent the actual obtained coverage values and an array showing the average coverage values. To compute the average branch (respectively, method, line, or instruction) coverage achieved for a given library, the number of all covered branches (respectively, methods, lines, or instructions) in all executions is summed and divided by the total number of branches in the library multiplied by the number of executions (20).

To identify the origin of any observed differences between JTExpert and EvoSuite, we minutely analyze the classes wherein a significant difference of coverage is observed: for each library, we create an Eclipse[4] project containing the classes under test and the generated test

[4] `https://www.eclipse.org`

suites. Then we observe the branches covered by one tool and missed by the other using the plugin EclEmma[5]. We make our interpretations according to the type of branches at the root of the differences.

We also perform statistical tests on the coverage results. We choose Mann-Whitney U-test Mann et Whitney (1947) and Vargha-Delaney's $\widehat{A}_{12}$ effect size measure (Vargha et Delaney, 2000). Both measures are recommended as statistical tests to assess whether a novel random testing technique is indeed useful (Arcuri et Briand, 2014).

- To assess the statistical significance of the average difference between JTExpert's results and EvoSuite's, Mann-Whitney U-test Mann et Whitney (1947) is used and the p-values are computed. Generally, the U-test is applied to compare whether the average difference between two groups is really significant or if it is due to random chance. The average difference between two groups is considered statistically significant if its p-value is less than the traditional definition of significance (0.05). Because we make multiple comparisons (20) according to Bonferroni adjustments the new threshold of significance become $\frac{0.05}{20}$, or 0.0025 (Weisstein, 2014);
- The U-test may be statistically significant but the probability for JTExpert to outperform EvoSuite may be small. To assess this probability, we also compute Vargha-Delaney's $\widehat{A}_{12}$ effect size measure Vargha et Delaney (2000). The $\widehat{A}_{12}$ measures the probability that JTExpert yields higher code coverage than EvoSuite, e.g., $\widehat{A}_{12} = 0.9$ means JTExpert would obtain better results than EvoSuite 90% of the time.

Understanding JTExpert Behavior

To gain a better understanding of the behavior of our approach and the contribution of each proposed component in its results, we carry out several different sets of experiments, each one focusing on one of the proposed components. First, we analyze JTExpert without any of its components (`JTE-All`) except a basic instance generator and the random search that targets all branches at the same time. Then, for a given component (e.g., problem representation, seeding strategy, diversification strategy) we analyze JTExpert coverage over time in the absence of this component, its contribution being measured by the difference observed between JTExpert and the version of JTExpert without this component.

Because we use JaCoCo to measure coverage, we can not get the progress of coverage over time, so to get this information, we perform a set of experimentations: an experimentation is performed for each search time in the set {10s, 30s, 50s, 100s, 150s, 200s}. Each experi-

[5]EclEmma is based on the JaCoCo code coverage library available at: `https://www.eclipse.org`

mentation was performed with the same conditions as the first one (20 executions for each unit under test). The average branch coverage in terms of time is analyzed.

8.3.2 Results

All graphics and statistical analyses presented in this study are performed with R[6] version 3.1.0 (Team *et al.*, 2012).

8.3.3 Comparing JTExpert to EvoSuite

Figures 8.3a, 8.3b, 8.3c, 8.3d, and 8.3e report the box-plots of the achieved coverage in 200 seconds per CUT by each tool on the Joda-Time, Barbecue, Commons-lang, and Lucene library, and all libraries together, respectively. Each box-plot compares JTExpert to EvoSuite using the four metrics of code coverage offered by JaCoCo: methods, branches, lines, and instructions.

Table 8.2 summarizes the results in terms of average coverage achieved by each tool at 200 s for each CUT. It shows the average of the four metrics. Triangles directed upward show the points of comparison wherein JTExpert outperforms EvoSuite by a percentage ranging from 2.5 % to 24 % while a square represents a difference of less than 2.5 %. Table 8.3 reports the $\widehat{A}_{12}$ effect size measure as well as the p-values of the statistical U-test.

The benchmark Vos (2013) contains two other classes from the library Sqlsheet but we do not report the results of these two classes because neither JTExpert nor Evosuite could generate any test data. The challenge in these classes is generating an instance of the CUT. These classes offer a constructor that needs two parameters: one is a URL and the other is a string. The first parameter must be a URL referencing an existing Microsoft Excel file and the second parameter must hold the name of an existing sheet in the file. The likelihood of randomly instantiating this type of class is almost null. Hence, to automatically generate an instance for such a class, the instance generator must understand the context of use of the class, which we will tackle in our future work.

A glance at Figures 8.3a, 8.3b, 8.3c, 8.3d, 8.3e and Table 8.2 shows that JTExpert outperforms EvoSuite by covering test targets that EvoSuite failed to cover. Among 20 comparisons, EvoSuite achieved almost the same performance as JTExpert for instruction coverage on Barbecue whereas JTExpert is dominant for all other metrics and libraries. The difference over EvoSuite reached 23.91 % using the metric branch coverage on the library Commons-lang. In total, JTExpert covers on average 3,952 instructions more than EvoSuite.

[6]Available at http://www.R-project.org

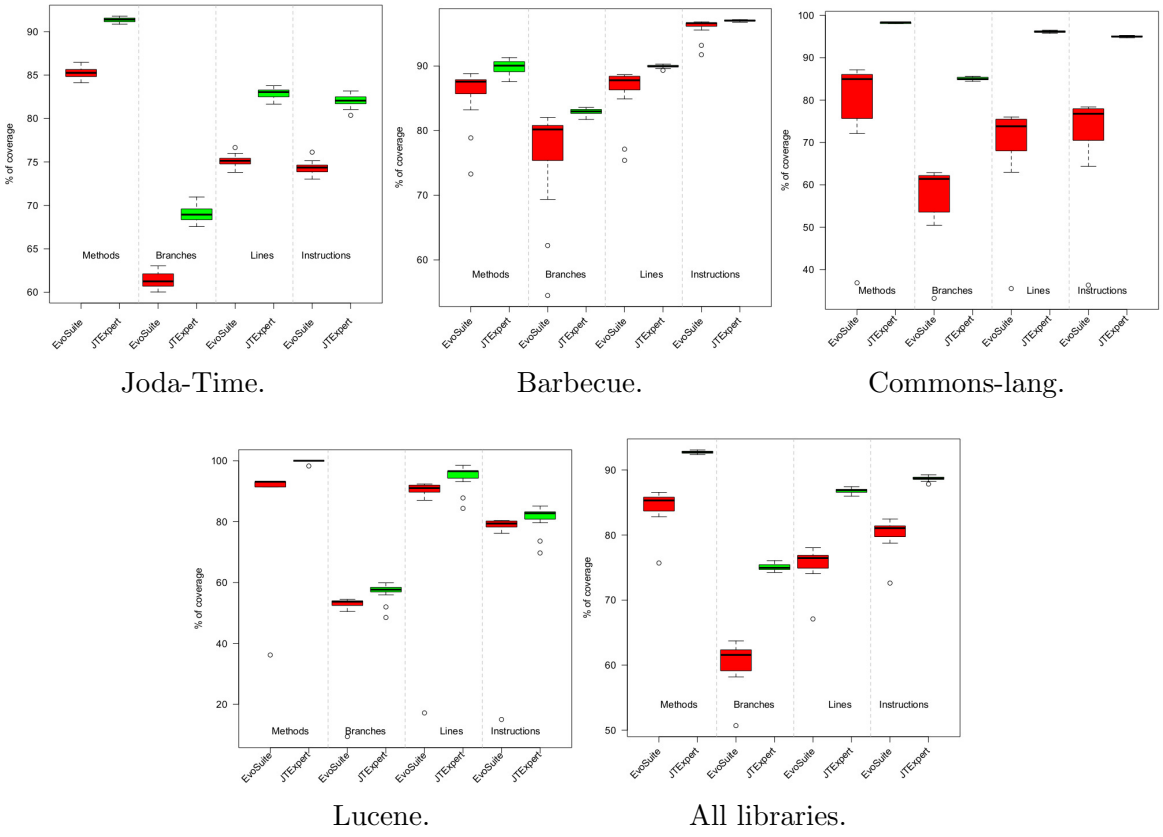

Figure 8.3: Comparison of JTExpert and EvoSuite on all libraries in terms of method coverage, branch coverage, line coverage, and instruction coverage.

In Table 8.3, all U-test's p-values are less than 10^{-4} thus less than the threshold of significance 0.0025. To say that there is a difference, in all cases, is taking less than 20×10^{-4} percent risk of being wrong. Thus, we conclude that there is a statistically significant difference between JTExpert's results and EvoSuite's. Also, almost all $\widehat{A}_{12}$ effect size measure values are equal to 1. Therefore, JTExpert is practically certain to achieve a higher code coverage than EvoSuite. Even when $\widehat{A}_{12}$ is less than 1, there is a high probability (at least equal to 0.88) for JTExpert to achieve better coverage than EvoSuite.

To summarize, box-plots, average code coverage, Mann-Whitney U-test, and Vargha-Delaney's $\widehat{A}_{12}$ effect size measure results support the superiority of our approach over EvoSuite in terms of code coverage.

8.3.4 Comparing JTExpert and EvoSuite in Details

To better understand the origin of the observed differences in terms of code coverage between JTExpert and EvoSuite, we analyze the code coverage at the class level. We would have

Table 8.2: Summary of the experimental results. Comparison with EvoSuite in terms of average coverage.

Libraries	**Tools**	**% of average coverage in terms of**			
		Methods	**Branches**	**Lines**	**Instructions**
Joda-Time	EvoSuite	84.92	60.82	74.80	73.94
	JTExpert	▲ 91.33	▲ 69.01	▲ 82.90	▲ 82.09
Barbecue	EvoSuite	86.80	77.49	87.38	96.40
	JTExpert	▲ 89.75	▲ 82.89	▲ 89.94	■ 97.02
Commons-lang	EvoSuite	83.72	61.15	73.54	76.12
	JTExpert	▲ 98.27	▲ 85.06	▲ 96.17	▲ 95.00
Lucene	EvoSuite	92.58	52.40	90.26	78.59
	JTExpert	▲ 99.91	▲ 57.15	▲ 95.17	▲ 81.53
All	EvoSuite	**85.08**	**61.61**	**76.26**	**80.92**
	JTExpert	▲ **92.73**	▲ **75.04**	▲ **86.82**	▲ **88.70**
Diff.	%	**+7.65**	**+13.43**	**+10.56**	**+7.78**
	#	**+152.85**	**+741.20**	**+983.45**	**+3,952.16**

liked to automatically analyze the lower levels (e.g., methods, branches, statements) but the JaCoCo reports do not offer us this information. Thus, we manually analyze and interpret the code coverage on lower levels.

Figure 8.4 presents the box-plots of the branch coverage achieved by each tool on the classes where a significant difference is observed. The coverage achieved on a given class is compared between two vertical lines. Each comparison contains a box-plot for EvoSuite's coverage, a box-plot for JTEpert's coverage, the name of the class, and the total number of branches in that class written in brackets. In Figure 8.4, JTExpert has higher coverage than EvoSuite's on the first fourteen classes, from the left, that represent 60% of the total number of branches under test. EvoSuite has higher coverage than JTExpert on the last seven classes that represent 5.6% of the total number of branches under test. Evosuite achieved better coverage in some small or medium size classes whereas JTExpert has higher coverage in some other small or medium size classes and is dominant on large classes. Overall, Figure 8.4, supports the observation that JTExpert is more effective than EvoSuite on large classes. This observation can be explained by the complexity of identifying a required sequence of methods to reach a test target in a large class, i.e., a significant number of sub-classes and methods substantially decreases the likelihood to get a relevant sequence without a static analysis. The proposed problem presentation helps JTExpert to reach more test targets efficiently by trying only relevant sequences of method calls, whereas EvoSuite may try a

Table 8.3: Results of computing U-test and the $\widehat{A}_{12}$ effect size measure on JTExpert's results compared to EvoSuite's.

Libraries	**Comparing JTExpert to EvoSuite in terms of average coverage**							
	Methods		**Branches**		**Lines**		**Instructions**	
	U-test (p)	$\widehat{A}_{12}$	**U-test (p)**	$\widehat{A}_{12}$	U-test (p)	$\widehat{A}_{12}$	**U-test (p)**	$\widehat{A}_{12}$
Joda-Time	6.71e-08	1	6.76e-08	1	6.78e-08	1	1.45e-11	1
Barbecue	3.54e-07	0.96	6.95e-08	0.99	6.32e-08	1	1.05e-07	0.99
Commons-lang	3.27e-08	1	6.77e-08	1	6.67e-08	1	6.74e-08	1
Lucene	5.41e-09	1	8.98e-06	0.91	9.48e-06	0.90	3.88e-05	0.88
All	6.69e-08	1	1.45e-11	1	6.78e-08	1	6.79e-08	1

significant number of sequences of method calls without getting a good one.

Class `org.joda.time.format.ISOPeriodFormat` has five private data members and five public methods. Each method contains two branches and uses a different data member for its conditional statement. Figure 8.5 presents the source code of one of the methods. The data member `cAlternateWithWeeks` is used and modified only by this method. Hence, to reach both branches, the method must be called twice in a same sequence. All the other methods follow same pattern. The twenty test suites generated by EvoSuite cover only the five branches that require a null data member, whereas those generated by JTExpert cover all ten branches, thanks to its problem representation that allows JTExpert to understand that reaching those branches requires two calls of the methods.

Class `net.sourceforge.barbecue.CompositeModule` has a private data member, `modules`, and contains three loops over `modules` in three different methods. Figure 8.6 presents a part of the source code of `CompositeModule`. To enter inside a loop, the list `modules` must contain at least one item. Hence, to reach a given loop, the method `add(Module module)` must be called before any method containing a loop. The twenty test suites generated with EvoSuite could not enter in any loop, whereas those generated by JTExpert cover two of the three loops. JTExpert missed covering the third loop because it is inside a protected method. JTExpert's problem representation makes the difference.

In package `org.joda.time.format`, classes `DateTimeFormatterBuilder` and `PeriodFormatterBuilder` contain seventeen class members: ten in `DateTimeFormatterBuilder` and seven in `PeriodFormatterBuilder`. The classes contain 144 methods: 92 in `DateTimeFormatterBuilder` and 52 in `PeriodFormatterBuilder`. EvoSuite has difficulty in reaching methods declared in class members: the test suites generated with EvoSuite could not reach 44 methods whereas the test suites generated with JTExpert missed only 19 methods. The

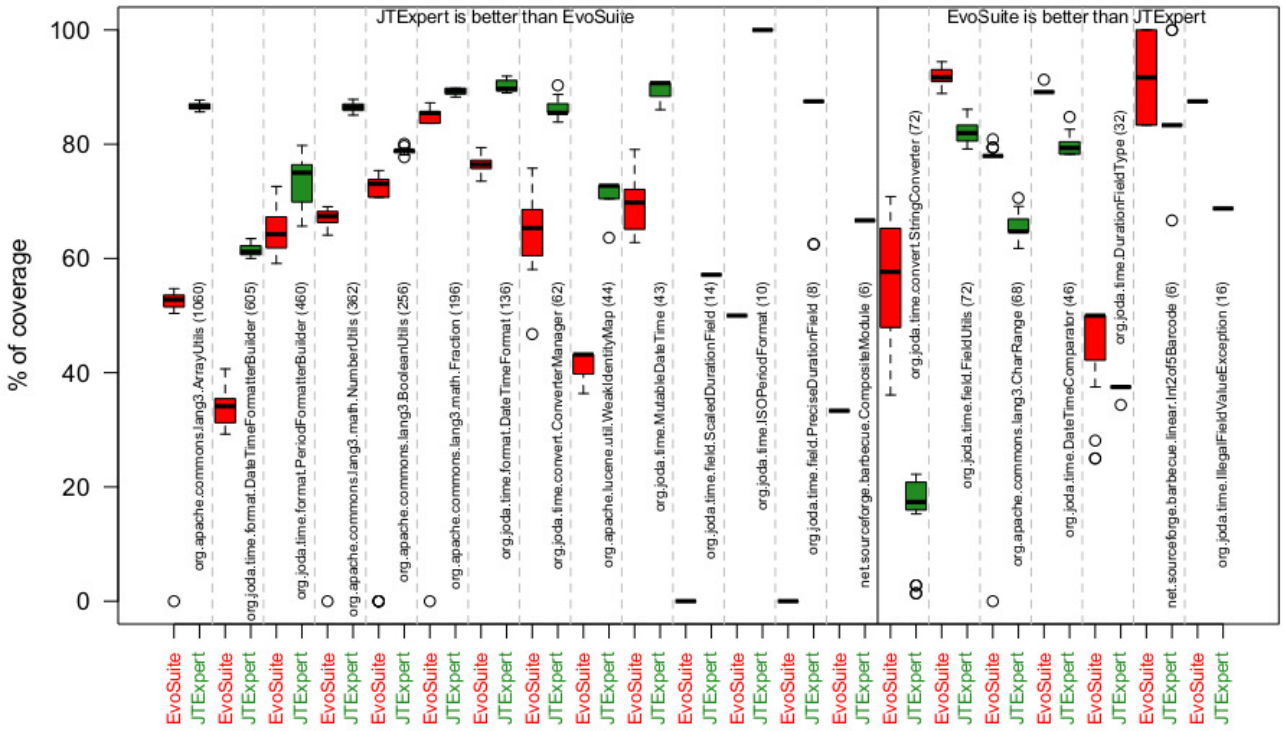

Figure 8.4: Comparison of JTExpert and EvoSuite on classes in terms of branch coverage.

19 methods missed by JTExpert are essentially in four class members: `DateTimeFormatterBuilder$Fraction`, `DateTimeFormatterBuilder$UnpaddedNumber`, `DateTimeFormatterBuilder$FixedNumber`, and `PeriodFormatterBuilder$Composite`. JTExpert was unable to instantiate the first three class members because they do not offer any accessible means-of-instantiation. EvoSuite could not or did not try to instantiate class members and reached a weak coverage compared to JTExpert. Its problem representation and means-of-instantiations make JTExpert more effective than EvoSuite by buildings more relevant sequences of methods and means-of-instantiations to reach methods in class members.

As shown in Figure 8.1, class `org.apache.lucene.util.WeakIdentityMap` contains an anonymous class with two private data members, a private method, and three public methods. Because there is no call to the three public methods, only a direct call can reach them. Also, branches in the anonymous class depend on the state of its enclosing class, `WeakIdentityMap`. Hence, to reach the methods in the anonymous class, an instance of this class is required in a

```
1   public static PeriodFormatter alternateWithWeeks() {
2       if (cAlternateWithWeeks == null) {
3           cAlternateWithWeeks = new PeriodFormatterBuilder()
4               .appendLiteral("P")
5               .printZeroAlways()
6               .minimumPrintedDigits(4)
7               .appendYears()
8               .minimumPrintedDigits(2)
9               .appendPrefix("W")
10              .appendWeeks()
11              .appendDays()
12              .appendSeparatorIfFieldsAfter("T")
13              .appendHours()
14              .appendMinutes()
15              .appendSecondsWithOptionalMillis()
16              .toFormatter();
17      }
18      return cAlternateWithWeeks;
19  }
```

Figure 8.5: Source code of method `org.joda.time.format.ISOPeriodFormat. alternateWithWeeks()`

relevant state. The twenty test suites generated by EvoSuite reached only two methods out of four and could not cover any branch, whereas those generated by JTExpert reached the four methods and covered 12 branches out of 16. Thanks to the means-of-instantiations that allow JTExpert to instantiate an anonymous class and with the help of the problem representation, JTExpert puts the instances of the class, `WeakIdentityMap` and the anonymous class in desired states to reach the methods and branches inside the anonymous class.

EvoSuite could not generate test data for two classes: `org.joda.time.field. ScaledDurationField` and `org.joda.time.field.PreciseDurationField`. The source code of EvoSuite is not available, so we cannot know the reason for this behavior but it may be due

```
1   public int widthInBars() {
2       int width = 0;
3       for (Iterator iterator = modules.iterator(); iterator.hasNext();) {
4           Module module = (Module) iterator.next();
5           width += module.widthInBars();
6       }
7       return width;
8   }
9
10  public void add(Module module) {
11      modules.add(module);
12  }
```

Figure 8.6: Part of the source code of class `net.sourceforge.barbecue. CompositeModule`

to its inability to generate an instance of the CUT. Consequently, JTExpert outperforms EvoSuite by covering eight out of 14 branches in `ScaledDurationField` and seven out of eight in `PreciseDurationField`.

In class `org.joda.time.convert.ConverterManager`, on average, each test suite generated by EvoSuite missed 13 branches compared to a test suite generated by JTExpert. The 20 test suites generated by EvoSuite missed four branches that require a `null` object (e.g., `if(object==null)`). The 20 test suites generated by EvoSuite, on average, covered only one of those four branches, whereas the test suites generated by JTExpert covered all four branches. The seeding of a `null` constant benefits JTExpert.

Three classes from the library Commons-lang, `ArrayUtils`, `BooleanUtils`, and `NumberUtils`, contain a significant number of conditional statements to check `null` objects and arrays of lengths equal to 0. Class `ArrayUtils` contains 170 branches requiring a `null` object and 60 branches requiring an array of length 0. Class `BooleanUtils` contains 26 branches requiring a `null` object and six branches requiring an array of length 0. Class `NumberUtils` contains 30 branches requiring a `null` object and 12 branches requiring an array of length 0. The 20 test suites generated by EvoSuite reached 31 branches requiring a `null` object: 10 branches in `ArrayUtils`, 14 branches in `BooleanUtils`, and seven branches in `NumberUtils`. These test suites could not reach the branches that required an array of length 0. In contrast, the test suites generated by JTExpert covered all branches requiring a `null` object or an array of length 0. The generator of containers and the seeding of `null` constants benefits JTExpert. The strategy used in EvoSuite to seed `null` constants is not enough and seeding the constant `null` with a constant probability equal to 5% is necessary to reach all branches involving that constant. Containers should be generated in a different way than other objects as proposed in Section 8.2.2.

Class `org.joda.time.MutableDateTime` extends class `BaseDateTime` and defines 103 methods that are mostly setters that take integers as parameters. In class `MutableDateTime`, there are no visible preconditions on the parameters but the parameters must satisfy the preconditions defined in the superclass, e.g., a value must be an hour of the day in the range [0,23] or a day of the week in the range [1,7]. These "hidden" preconditions make the task of covering methods in `MutableDateTime` harder. The test suites generated by EvoSuite could not cover 14 methods, whereas the test suites generated by JTExpert missed only four methods. This difference comes from the seeding strategy and the search heuristic. The seeding strategy offers adequate values that satisfy preconditions in the superclass but the way each tool uses them is different: EvoSuite relies on a genetic algorithm. If its initial population lacks values that satisfy the hidden preconditions, then EvoSuite may make a lot of attempts using the same population before introducing new values through its mutation

operator, whereas the random search in JTExpert uses different values. Further, the absence of preconditions may make the GA blind and worst than a guided random search. Therefore, the seeding strategy combined with the random search is at the root of the observed difference between JTExpert and EvoSuite on class `MutableDateTime`.

Classes `org.joda.time.field.FieldUtils` and `org.joda.time.DateTimeComparator` contain branches and return statements that rely on disjunctive and conjunctive conditions. For such statements, the number of branches at the Java Bytecode level is different from at the Java source code level. For example, to cover all branches in `boolean foo(boolean a, boolean b){return a && b;}`, generates only one test data, whereas this function contains four branches at the bytecode level. Thus JaCoCo favors EvoSuite that works with bytecode and penalizes JTExpert. Consequently, EvoSuite reaches seven branches in class `FieldUtils` and four branches in class `DateTimeComparator` more than JTExpert.

Class `org.joda.time.convert. StringConverter` contains 72 branches, 42 of which are in method `getDurationMillis(Object object)` in which all the branches depend on the return value of `object.toString()`: To reach more than 28 branches, the method `object.toString()` must return a string that starts with the substring `PT` and finished with `S`, i.e., gets the value as a string in the ISO8601 duration format. For example, "PT6H3M7S" represents 6 hours, 3 minutes, 7 seconds. The method `object.toString()` must match the `toString()` method of the class `ReadableDuration`. In the 20 test suites generated, JTExpert failed to generate such an object, whereas EvoSuite generated the required objects. Consequently, EvoSuite outperforms JTExpert on class `StringConverter` with 29 branches. After inspecting JTExpert source code, we found that we restricted the list of stubs to instantiate class `java.lang.Object` to some classes, e.g., Integer, String, and the CUT. Thus, JTExpert could not generate an instance of a class that implements the interface `ReadableDuration`. This limitation may be fixed by enlarging the list of stubs of the class `java.lang.Object` to contain any other class that was instantiated during the search.

To summarize, the sample of classes analyzed above is representative because it covers classes in which we observed differences between the two approaches. This analysis shows the actual effects of the proposed components and clarifies the advantages and weaknesses of the compared approaches. It also shows the types of test target for which JTExpert outperforms Evosuite: branches involving a data member, branches requiring the `null` constant, branches requiring a container or string with length equal to 0, methods that are declared inside classes members or anonymous classes, and methods that contains hidden preconditions. Also, it reveals the limitation in our generator for class `java.lang.Object`.

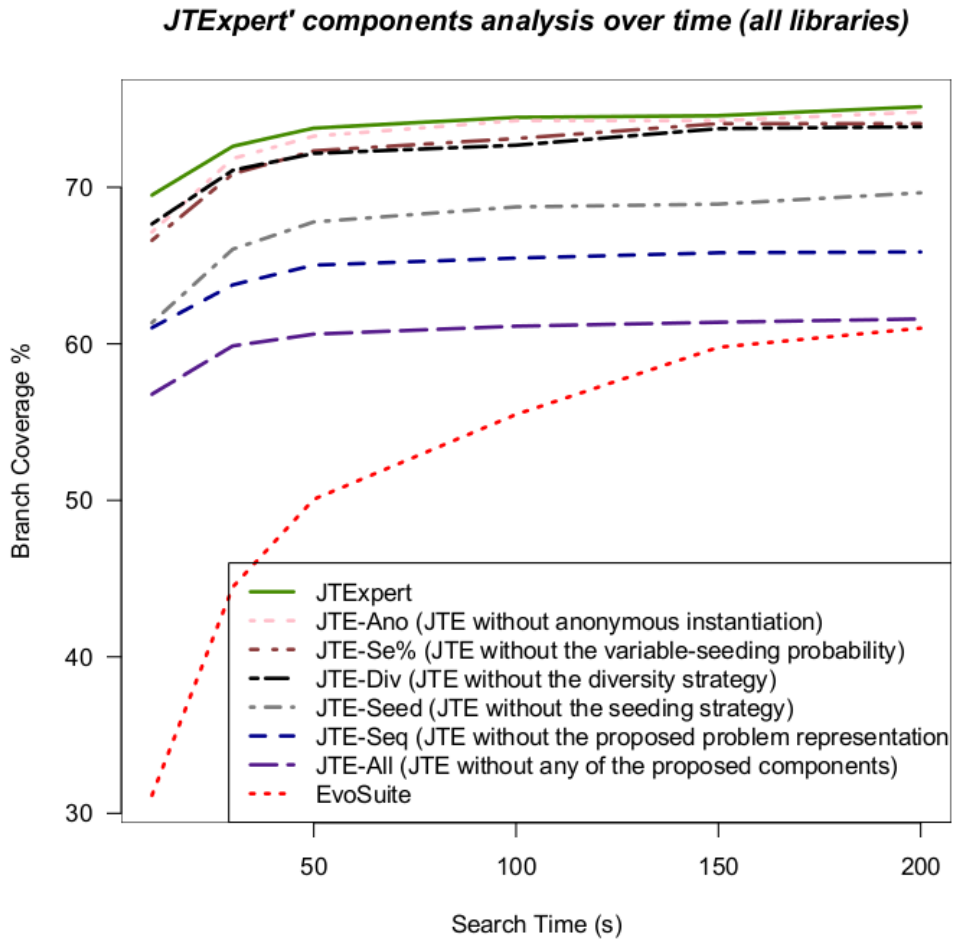

Figure 8.7: Contribution of each proposed component in terms of average branch coverage over time.

8.3.5 Understanding JTExpert behavior

We now quantify and measure the contribution of each component. Figure 8.7 shows the difference in terms of average branch coverage between JTExpert and six other versions wherein at least one proposed component is disabled: JTExpert without all components (JTE-All), JTExpert without the generator of sequences of method calls (JTE-Seq), JTExpert without the seeding strategy (JTE-Seed), JTExpert without the variable probability in the seeding strategy (JTE-Se%), JTExpert without the diversification strategy (JTE-Div), and JTExpert without the instantiation of anonymous classes (JTE-Ano).

Figure 8.7 reflects the achieved results in terms of average branch coverage for all classes. It shows that JTExpert performs better than Evosuite in terms of efficiency as well. JTExpert is more efficient because, at 10 seconds, it reaches a high branch coverage, almost 70 %,

whereas Evosuite reaches 31 % coverage at 10 seconds and 61 % at 200 seconds. There is a large difference in terms of time required to reach the same level of coverage because Evosuite does not reduce the search domain of D2 and D3 and because it does not have a diversification strategy, its seeding strategy uses a fixed seeding probability, and it covers only primitive types and strings. Also, the random search implemented in JTExpert does not waste time with complex branches. Furthermore, the state-modifier methods, target-viewfinder methods, and the instance generator guide the random search to quickly reach less complex branches.

In a first experimentation, `JTE-All`, each proposed component was disabled or replaced with a simple version as explained in the next paragraphs. `JTE-All` performed better than EvoSuite, especially on the Common-lang library and but is less effectiveness on Barbecue. For the two other libraries, Joda-Time and Lucene, EvoSuite performed better than `JTE-All`.

On all libraries, `JTE-All` is more efficient than EvoSuite because at 10 s it reached 56% branch coverage, whereas EvoSuite reach only 31%. This difference can be explain by the different search heuristics implemented in `JTE-All` and EvoSuite: If EvoSuite generates an initial population that lacks some methods leading to some easy branches, then it must wait until the mutation operator injects them, whereas the random search used in `JTE-All` is more likely to quickly select these methods in a sequence, hence it reaches more branches early.

At 200 s, EvoSuite (genetic algorithm) is supposed to outperform `JTE-All` (random search) but we observed almost the same branch coverage. To understand this behavior, we analyzed the branches reached on each class. Figure 8.8 presents the box-plots of the achieved branch coverage with `JTE-All` and EvoSuite at 200 s on the classes where a significant difference was observed.

EvoSuite outperforms `JTE-All` on classes from different libraries, particularity Joda-time. In these classes, branches are complex to reach because they are embedded in sub-classes or private methods, e.g., `org.joda.time.format. DateTimeFormatterBuilder` contains 14 sub-classes. EvoSuite benefits from genetic algorithm to reach a complex test targets compared to a random search.

JTE-All outperforms EvoSuite on three large classes in Commons-lang. Most of the methods in these classes take containers as parameters: 208 methods in class `ArrayUtils` and 13 in class `NumberUtils` use arrays. Also, the difficulty to reach branches in these classes lies in the generation of diversified instances of the methods' parameters. For example, a same method may contain a condition requiring a null array, a condition requiring an array with length equal to 0, and a condition requiring an array containing many elements. To cover branches in such a method, it is enough to call it with different types of array, i.e., to have

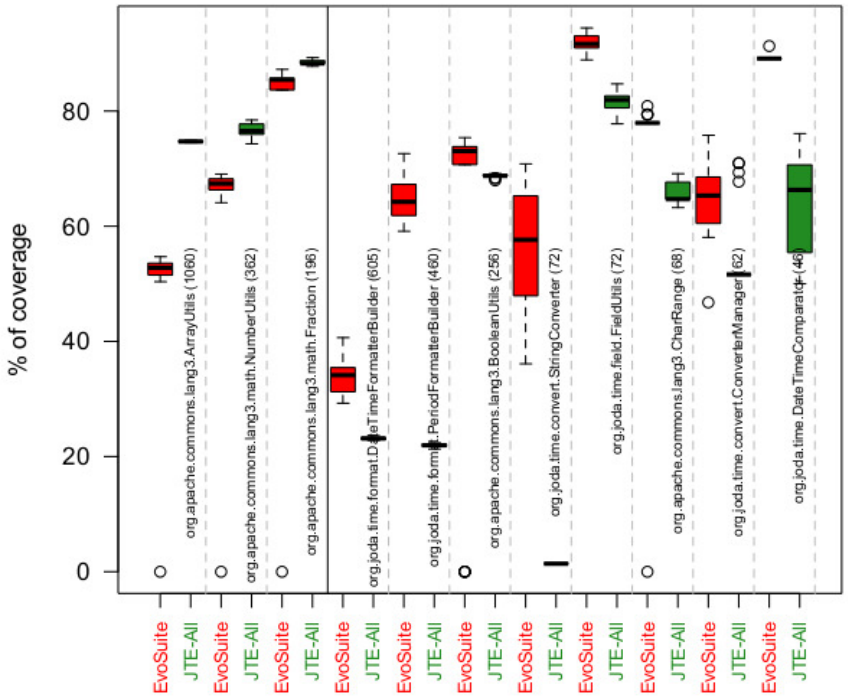

Figure 8.8: Comparison of JTE-All and EvoSuite on classes in terms of branch coverage at 200 s.

an array generator that generates diversified types of array.

We think that we could not see the difference between `JTE-All` and EvoSuite at 200 s because the containers generator implemented in `JTE-All` hides the advantage of the genetic algorithm implemented in EvoSuite on complex branches.

In a second experimentation, `JTE-Seq`, the generator of sequences of method calls was replaced by a random generation of sequences of method calls. The comparison to JTExpert in Figure 8.7 shows that the use of the proposed representation is beneficial for the search, with an average branch coverage equal to 9.28 % at 200 s, where the average branch coverage increased from 65.87 % to 75.15 %. The improvement is significant for the classes in Joda-Time and Lucene: the average branch coverage for Joda-Time increases from 53.55 % to 69.01 % and for Lucene, from 47.20 % to 57.15 %. For the other libraries, the obtained results are almost the same, i.e., there are no improvements. The better performance obtained on Joda-Time and Lucene and not on Commons-lang and Barbecue can be explained by the complex class structures found in the first two: in Joda-Time and Lucene, one Java file

or class defines more than one class (e.g., the Java file `DateTimeFormatterBuilder.java` defines 15 classes). A CUT that contains nested classes (e.g., member class, local class, or anonymous class) needs guidance during the test-data generation. We show that the proposed representation improves coverage, particularly for complex classes under test.

In a third experimentation, `JTE-Ano`, anonymous classes are tested through their enclosing class, i.e., the closest accessible class that contains the anonymous CUT. The comparison to JTExpert in Figure 8.7 shows that the proposed means-of-instantiations of anonymous classes slightly enhances the search process, with an average branch coverage equal to 0.34 % at 200 s. This improvement may seem insignificant but such magnitude on all classes was expected because such anonymous classes are rare in the chosen libraries. To estimate the actual improvement that the instantiation of anonymous classes may bring, we analyzed class `org.apache.lucene.util.WeakIdentityMap` separately. We found that the proposed means-of-instantiations of anonymous class enhance significantly the average branch coverage of this class, where the coverage increased from 39.77 % (`JTE-Ano`) to 71.47 % (JTExpert). This is a large enhancement in terms of branch coverage because the anonymous class defined in `org.apache.lucene.util.WeakIdentityMap` contains a significant number of branches.

In a fourth experimentation, `JTE-Div`, the diversification strategy was replaced by a random selection of means-of-instantiations. The comparison to JTExpert in Figure 8.7 shows that the proposed diversification strategy enhances the search process, with an increase of average branch coverage equal to 1.28 % at 200 s, where the average branch coverage increased from 73.87 % to 75.15 %. At first glance, this improvement seems insignificant. However, this magnitude is expected because of the small number of classes that are hard to instantiate. For example, class `org.apache.lucene.util.FixedBitSet` is hard to instantiate because calling some of its constructors with random arguments may hang the test data generation process. Thus, the diversification strategy is beneficial to the search, with an average branch coverage equal to 26.53 %, where the coverage on this particular class passed from 26.09 % to 52.62%. We show that the proposed diversification strategy of means-of-instantiations improves coverage, especially for classes that defines some means-of-instantiations that may be harmful with randomly generated arguments.

translated into many conditional statementsIn a fifth experimentation, `JTE-Seed`, the proposed seeding enhancements were simply disabled, i.e., without seeding `null` and with a constant seeding probability equals to 0.2. Thus, the seeding strategy used in this experimentation is equivalent to one studied in Fraser et Arcuri (2012). The comparison to JTExpert in Figure 8.7 shows that seeding `null` and using a variable seeding probability are beneficial for the search, with an increase of average branch coverage equal to 5.60 % at 200 s, where the average branch coverage increased from 69.65 % to 75.15 %. In this 5.60 % enhancement, the

variable probability of seeding contributes by 1.08 % as shown in the graph `JTE-Se%` in Figure 8.7. The improvement is significant for the classes in Commons-lang and Lucene, with an enhancement almost equal to 10 %, where the average branch coverage increased from 75.63 % to 85.06 % on Commons-lang and from 44.80 % to 57.15 % on Lucene. On Joda-Time and Barbecue, the improvement is less significant, below 3 %. The significant enhancement on the first two libraries can be explained by the significant number of branches that rely on null pointer checks. The reason for the small improvement observed on Joda-Time and Barbecue is the small number of conditions that they use to prevent null pointer exception. During the execution of the test suites generated for these two libraries, we observed that a significant number of null pointer exceptions occurred. Seeding `null` may be also beneficial to raise null pointer exceptions. We conclude that seeding `null` improves the coverage, especially for classes that systematically check for null pointers before using the instance of a class.

To summarize, every proposed component improves the coverage for all or some cases of the CUT. The large difference in terms of code coverage achieved over EvoSuite comes from the accumulated contributions of all proposed components.

8.3.6 Threats to Validity

The results showed that using JTExpert to generate test data, improves SBST performance in terms of runtime and code coverage. Yet, several threats potentially impact the validity of the results of our empirical study. We discuss in details these threads on our results, following the guidelines provided in Yin (2014).

Construct validity threats concern the relationship between theory and observation. They are mainly due to errors introduced in measurements. In our study, they related to the measure of the performance of a testing technique. We compared JTExpert and EvoSuite in terms of coverage, which is widely used to measure the performance of a testing technique. As a second measure of performance, we chose time rather than the number of fitness evaluations because a fitness evaluation has a different meaning in the two tools: in EvoSuite, a chromosome is a set of test-data candidates and one evaluation may require the execution of many test-data candidates Fraser et Arcuri (2013c) whereas in JTExpert, we evaluate each test-data candidate separately. Moreover, time is the most important constraint in the testing phase of an industrial system Ciupa *et al.* (2008).

Internal validity threats arise from the empirical study process methodology. A potential threat comes from the natural behavior of any search-based approach: the random aspect in the observed values, which may influence the internal validity of the experiments. In general, to overcome this problem, the approach should be applied multiple times on samples with a reasonable size an statistical tests should be used to estimate the probability of mistakenly

drawing some conclusions. In our empirical study, each experiment took 200 s and was repeated 20 times. The libraries contain in total 50,799 instructions and 5,519 branches. Also, we computed Mann-Whitney U-tests and evaluated Vargha-Delaney's $\widehat{A}_{12}$ effect size measure. Therefore, experiments used a reasonable size of data from which we can draw some conclusions.

External validity threats concern the possibility to generalize our observations. A potential threat is the selection of the libraries and classes used in the empirical study. All these libraries and classes have been used to evaluate different structural testing approaches in the past Vos (2013); Fraser et Zeller (2012), thus they are a good benchmark for evaluating our approach.

Another potential threat comes from the EvoSuite parameters: we did not try different combinations of parameters values to show empirically that our approach is robust to EvoSuite parameters. However, according to Arcuri and Fraser Arcuri et Fraser (2011), the default configuration of EvoSuite performs well, but it may not be as good as a specific configuration for each class. Because it is hard and impractical to find a best configuration for each class, the default parameter settings of EvoSuite can be considered as a good practical configuration.

Another thread is the choice of Java to implement our approach, which could potentially affect its external validity. We implemented our prototype, JTExpert, to generate test data for classes written in Java, although the presented components can be adapted to any OOP language, e.g., C++. The choice of Java is technical because: (1) we believe that the best programming language is the language that you master better, and in our laboratory Ptidej[7], we have an extensive experience in analyzing Java source code; (2) there are many available open-source tools (e.g., JDT, Reflexions) that made our development task easier; (3) it is much easier to debug the tool in Java; (4) it is easier to get a correspondence between a CUT, its instrumented class, and its test-data suite. In the described implementation, we referred to some APIs, such as the meta-data analyzer, Java Reflection API Reflection (2013), and the AST generator, Eclipse JDT development tools (JDT), to ease our development task. In general, for many OOP languages, there exist different meta-data analyzers and AST generators. In the worst case, if the approach must be implemented for an OOP language for which there is no meta-data analyzer or no AST generator, then four additional analyzers must be developed: (1) to get all means-of-instantitions of a given class, (2) to get all branches callers of a given method, (3) to get all branches modifiers of a given data member, and (4) to extract constants from the source code.

Reliability validity threats concern the possibility of replicating our study. We attempted

[7]Ptidej: Pattern Trace Identification, Detection, and Enhancement in Java. Website: http://www.ptidej.net

to provide all the necessary details to replicate our study: analysis process is described in detail in Section 9.2 and all the classes tested in this study are publicly available Vos (2013). Moreover, all tools, JTExpert, EvoSuite, JaCoCo, Eclipse, EclEmma, and R, used in this study are publicly available.

8.4 Summary

In the last decade, search-based software testing (SBST) has been extensively applied to solve the problem of automated test-data generation for procedural programming as well as for object-oriented programming (OOP). Yet, test-data generation for OOP is challenging due to the features of OOP, e.g., abstraction, encapsulation, and visibility that prevent direct access to some parts of the source code.

This chapter introduced an approach for OOP software test-data generation for unit-class testing whose novelty relies on analyzing statically the internal structure of a class-under-test (CUT) to reduce the search space and on a diversification strategy and seeding strategy. The approach sees the test-data generation problem as facing three difficulties: ($D1$) obtaining an instance of a CUT and other required objects; ($D2$) finding an adequate sequence of method calls to put the instance of the class in a desired state; and, ($D3$) finding an adequate method to reach the test target through it.

To solve $D1$, an instance generator based on a diversification strategy, instantiation of anonymous classes, and a seeding strategy are proposed to boost the search. Instantiating a class using different means-of-instantiations according to their complexities is beneficial to the search with an increase of the average coverage equals to 1.28%, up to 26.53% in certain cases. The instantiation of anonymous classes brought an enhancement to the average coverage equal to 0.34% but up to 41% in the case of class `org.apache.lucene.util.WeakIdentityMap`. Using a variable seeding probability for primitive types and seeding the `null` value for classes while generating instances increase the average branch coverage by 5.60%.

To solve $D2$ and $D3$, only methods that may change an instance of the CUT state and methods that may reach the test target are explored. This leads to a restructuring of the test data generation problem that improved the branch coverage by 9.28%.

As a result, and contrary to earlier approaches that use a traditional representation of the problem and randomly generate instances of objects, our approach and its implementation JTExpert, find a test-data suite and achieve a high code coverage (70%) in less than 10 s. We showed on more than a hundred classes taken from different open-source Java libraries that JTExpert has a higher code coverage than EvoSuite and needs less time.

JTExpert currently relies on a guided random search and supports branch coverage. We are working on adding further search algorithms, such as genetic algorithms and hill climbing.

Also we are focusing on three research directions: (1) enlarging the scope of JTExpert to support other structural testing criteria, such as data-flow coverage or mutation-coverage; (2) enhancing the instances generator to generate instances of classes that require understanding the context of their use; and, (3) studying the impact of seeding `null` with the instances of objects on the generation of null pointer exceptions.

CHAPTER 9

JTEXPERT: A LARGE SCALE EXPERIMENTATION

Exceptions are a subclass of software failures that may be results of erroneous inputs, hardware faults or logical errors in the software code Tracey *et al.* (2000). An unhandled exception in a software may generate undesirable behavior or lead to catastrophic consequences. Yet generating test data to raise exceptions during the software testing phase is a challenging task. Typically, it is easier to generate a test datum that covers a test target than generating a test datum that raises an exception in that test target. Only a small fraction of the large body of work on test-data generation are focusing on exceptions raising Tracey *et al.* (2000); Bhattacharya *et al.* (2011); Csallner et Smaragdakis (2004); Fraser et Arcuri (2013a).

To evaluate the scalability of our approach IG-PR-IOOCC and its implementation JTExpert on large software, this chapter presents an adaptation of our Java tool, JTExpert, to raise exceptions via branch coverage, i.e., an Exceptions-oriented Test-data Generation Approach. Then, it presents the experimental results of applying JTExpert on a large open-source software, the Apache Hadoop System. The results reveal 4,323 failures, two of them leading to JVM crashes.

9.1 Exception-oriented Test-data Generation

9.1.1 Motivating Example

The Java function `foo` in Fig. 9.1 takes two integers `i` and `j` and an instance `a` of a class `A` as input parameters, then it calls the method `a.add(int)` using the ratio of `i` and `j` as parameter. Suppose that the function `foo` is under test and the test target is to cover all-branch. Any randomly generated test data can cover this objective, e.g., $t_1 : \langle 1, 2, newA() \rangle$. The problem with such a test data is its inability to detect unhandled exceptions: t_1 cannot reveal any unhandled exceptions in the source code of the function `foo`. Yet, this source code contains at least two unhandled exceptions, i.e., if we execute the function `foo` using `j` equals `0` then the Java Virtual Machine (JVM) will throws a `DivisionByZeroException`; if `a` equals `null` then the JVM will throw a `NullPointerException`. Such unhandled exceptions in a source code may lead to abnormal behavior. Therefore, it is important to generate a test data suite covering the test targets and unhandled exceptions. An approach that aims to generate test data for code coverage may improve the robustness of a program under test by covering some exceptions.

```
1  public void foo(int i,int j, A a) {
2
3
4
5
6          a.add(i/j);
7  }
```

Figure 9.1: A `foo` function

```
1  public void fooUExceptions(int i,int j, A a) {
2      try{
3          a.add(i/j); //branch 1
4      }catch(Throwable){
5          //branch 2
6      }
7  }
```

Figure 9.2: How we see the `foo` function

9.1.2 Principle

The assumption underlying our research work is that every branch in a CUT may contain some unhandled exceptions. Typically, our approach sees any branch as surrounded by a virtual `try/catch` and tries to cover both branches, try and catch. Fig. 9.2 shows how our approach sees a branch: the actual block in a branch is surrounded by an virtual `try/catch` from which a new branch is created. For example, to cover all-branches in the function `foo` our approach stops search after generating a test data that reaches Line 6 in Fig. 9.1 and does not raise any exception and another one that stops at Line 6 by raising an exception. If our approach gets the input data $\langle 1, 2, newA() \rangle$ that seems to cover all-branches in `foo`, then it goes on looking for a second test-data such as $\langle 1, 2, null \rangle$ to generate an exception.

9.1.3 Exceptions Classification and Analysis

Generally, any operating system (e.g., Lunix, MacOS) or programming language (e.g., C++, Java) is supposed to offer constructs for handling exceptions. In Java, exceptions are classified in three classes: Checked Exceptions, Unchecked Exceptions, and Errors. Checked Exceptions must all be handled in the source code and are checked by the compiler, e.g., `IOException`. A programmer is expected to check for these exceptions and handle them. Thus, it is unlikely to find unhandled checked exceptions in source code. Unchecked Exceptions can be prevented in the source code, for example to avoid a `NullPointerException`, a programmer can check the nullity of any object before using it. Typically, this class features the most unhandled exceptions in a source code. Errors indicate serious problems at runtime, e.g., a function going into infinite recursive calls will cause the JVM to raise a `StackOverflowError`.

In this work, we focus on unhandled exceptions and crashes of the JVM. Unhandled exceptions can be classified in two categories: expected exceptions and unexpected exceptions. Expected exceptions are the subset of unchecked exceptions existing inside the CUT. A static analysis can identify all statements that are concerned with this category of exceptions, transform them to new branches, and cover them as branches Fraser et Arcuri (2013a).

Unexpected exceptions are more difficult to raise than expected exceptions because we cannot predict if a statement can throw an unexpected exception. Unexpected exceptions are the set of errors, fatal errors, and exceptions thrown by a dependency of the CUT that are not managed in the source code.

9.1.4 Test-data Generation to Raise Unhandled Exceptions

We can generate a test datum to reach an expected exception that is transformed to a branch using SB-STDG and a search heuristic (e.g., Genetic Algorithm) McMinn (2004). However, if we suppose that any branch may contain an unknown unhandled exception, then a search heuristic will be insufficiently guided. Therefore, the random search implemented in JTExpert may be more adequate.

During the search, JTExpert distinguishes test-data candidates by monitoring the branches executed without considering thrown exceptions. Thus, an additional monitoring of raised exceptions is required. This monitoring is possible via an executor of test-data candidates that must catch all classes of exceptions and analyze them to filter unexpected exceptions. To qualify an exception as unhandled, its origin statement should not be an exceptions thrower, e.g., an exception explicitly thrown by an `assert` statement should not be considered as unhandled exception. Thus, patterns of exceptions thrower are a key to determine whether an exception is handled or unhandled.

9.2 Empirical Study

This section presents an evaluation of our approach IG-PR-IOOCC, its implementation JTExpert, and Exceptions-oriented Test-data Generation for raising unhandled exceptions described in Section 9.1. We investigate the advantages and limitations of our approach by applying our adapted JTExpert tool on the large, widely-used, and well-tested Apache Hadoop System. The Apache Hadoop framework allows the distributed processing of large data sets across clusters of computers using simple programming models. The Hadoop System is composed of four main modules: (1) Common is the common utilities that support other modules; (2) Distributed File System is a distributed file system that provides high-throughput access to application data; (3) YARN is a framework for job scheduling and cluster resource management; (4) MapReduce is a YARN-based system for parallel processing of large data sets. Each module contains sub-modules: in total Hadoop contains around 43 sub-modules, 3,545 Java files, 7,531 classes, 88,971 methods, 148,691 branches, 421,032 lines, and 1,698,650 statements. Most of the classes are complex for test-data generation because they have a number of features that make them challenging for test-data generation:

1. Running Hadoop is highly dependent on external configurations files: every Hadoop module requires a particular configuration file that specifies many properties of the execution environment. A missed or malformed property may prevent the instantiation of certain classes and decrease the coverage.

2. Hadoop contains a file system: randomly instantiating its classes and executing methods is a high-risk task that can result in undesirable effects, corrupted files, data loss, and even serious damages to the underlying operating system. Thus, the test operating system must be very restrictive in terms of access rights. A solution to avoid this problem is to create a user with very restrictive access, e.g., that can only read classes under test and dependencies, execute JTExpert, and write test data in a specific directory. However, such restrictions may reduce source-code coverage.

3. Hadoop is composed of different types of applications: servers applications, client applications, web applications, protocols, and a distributed file system. Testing this sort of applications requires the cooperation of some other applications. For example, let us suppose that the CUT is a server application and some of its private methods are executable only if that server gets a particular request from a client application via a particular socket (port). Such a scenario is hard to automatically generate because there is nothing in the source code of the server that points to a client class.

4. Hadoop is designed to run on many heterogeneous execution environments: different hardware, operating systems, network connections. Thus, many parts of its code can be run only on a particular environment. For example, the native Hadoop library does not work with Cygwin or the Mac OS X platforms. Because we cannot use all possible environments to generate test data, finding a test data is impossible for some parts of the source code.

Automatically generating test data to reach a high code coverage for the Apache Hadoop System is therefore a difficult task and we claim that redirecting a part of this effort to raise unhandled exceptions in that project may be more interesting and beneficial for the Apache Hadoop System.

For every Java class in Hadoop, we generated a complete test-data suite to cover all branches according to our Exception-oriented Test-data Generation technique implemented in our tool JTExpert. For each CUT, we allowed a maximum of 600 seconds for test-data search. To perform the experiment, we used a Oracle Grid Engine comprising 42 similar nodes, each of them equipped with 2-dual core CPU 2.1 GHZ, 5GB of memory, a Fedora core 13 x86_64 as OS, and JDK 7.

9.2.1 Unhandled Exceptions Revealed

Table 9.1 summarizes unhandled exceptions raised in Hadoop. The test-data suites raise 16,046 unhandled exceptions, 4,322 of them not redundant because they point to different lines in the source code. These unhandled exceptions affect 4,061 methods in 1,920 classes.

A large number of reported exceptions are unchecked exceptions. The `NullPointerException (NPE)` is the most thrown unchecked exception: 13,663 `NPE` affecting 3,604 different lines, 3,377 different methods, and 1,343 different classes. This exception is thrown when the JVM attempts to execute a method or to access a field of a null object. The test data suites threw many other unchecked exceptions: 658 `ClassCastException`, 326 `IllegalArgumentException`, 296 `ArrayIndexOutOfBoundsException`, and 492 other unchecked exceptions. Such exceptions could be avoided by using some preconditions on the concerned statements. For example, to avoid `NPE` a developer must ensure that an object is not null before using its methods or fields. Two reasons are behind the significant number of unchecked exceptions: (1) multiple unhandled exceptions may be linked to the same root cause, e.g., a static field assigned to null may be the root cause of hundreds of `NPE`; (2) developers tend to insert the preconditions only if necessary for a current use rather than systematically inserting preconditions on unchecked exceptions. Yet, such exceptions may harm evolution and future use of an affected class.

The number of errors alerted is less significant comparing to unchecked exceptions, but the former may be more serious. The test data suites threw four different types of errors: 602 `NoClassDefFoundError (NCDFE)`, 75 `UnsatisfiedLinkError`, 2 `StackOverflowError`, and 9 `Fatal Error`. The most thrown error is `NCDFE` that happens when JVM tries to load in a class and no definition of the class could be found. A `Fatal Error` is the less thrown error because it is difficult to raise, to reproduce, and to fix. It indicates that the JVM met

Table 9.1: Summary of Unhandled Exceptions Raised in Hadoop

Exception	**Alerted**	**Alerted in different**		
		Lines	**Methods**	**Classes**
java.lang.NullPointerException (NPE)	13,663	3,604	3,377	1,343
java.lang.ClassCastException (CCE)	658	213	206	165
java.lang.NoClassDefFoundError (NCDFE)	602	87	86	86
java.lang.IllegalArgumentException (IAE)	326	104	102	100
java.lang.ArrayIndexOutOfBoundsException (AIOOBE)	296	119	108	71
Fatal Errors	9	2	2	2
Other Exceptions	492	193	180	153
All Exceptions	**16,046**	**4,322**	**4,061**	**1,920**

a serious problem during the execution of a program. Because a fatal error forces the JVM to quit abnormally, a test data generator cannot automatically detect such an exception or create a test data to reproduce it. The best solution to reproduce a `Fatal Error` is keeping a copy of the last test-datum candidate executed in the disk and considering it as an additional test-data. To further understand and help to fix the 9 `Fatal Error` generated, we manually analyzed their test data. We observed that most of them are directly or indirectly calling the method `org.apache.hadoop.io.WritableComparator.compareBytes(byte[] b1, int s1, int l1,byte[] b2, int s2, int l2)` using a `null` value for either `b1` or `b2`: we then concluded that this method is the root of one `Fatal Error`. Following the execution of this method, we found, in the class `org.apache.hadoop.io.FastByteComparisons`, an inappropriate instantiation of the class `sun.misc.Unsafe`, i.e., Hadoop developers have violated encapsulation rules by using reflection to get a private instance of the `Unsafe` class. The problem is calling the method `Unsafe.getLong` using a null object leads to JVM crashes due to an invalid memory access. The second `Fatal Error` points to the class `org.apache.hadoop.hdfs. ShortCircuitShm` that also violates encapsulation rules to call the method `Unsafe.getLongVolatile` with a null object, correspondingly the JVM crashes.

To confirm the reality of the raised exceptions, we randomly selected some exceptions (i.e., a number ranging between 10 and 30 of each type of exception), manually checked them and confirmed their existence. Also, we searched Hadoop's bugs and found that one of the two `Fatal Errors` has been already reported[1] to Apache HBase, which is based on Hadoop. The bug report and the related discussion link the bug to a byte comparator without specifying its origin. In contrast, we could find the root cause of that bug because we have different test data to reproduce it.

9.3 Summary

In this chapter, we presented an adaptation of our approach IG-PR-IOOCC and its implementation JTExpert to raise exceptions via branch coverage, i.e., an Exceptions-oriented Test-data Generation Approach. The results of applying JTExpert on the Apache Hadoop System revealed 4,323 distinct failures two of them leading to JVM crashes. We confirmed through a manual statistical check that all unhandled exceptions raised are real and there is no false alert injected by JTExpert. Based on a reported bug in the Apache HBase project, we could show that any raised exception can be met in a real application of Hadoop and a test data may significantly facilitate fixing a failure. Therefore, we can conclude that JTExpert scales well to large software and our approach, Exception-oriented Test-data Generation, is useful for raising unhandled exceptions and very helpful for fixing them.

[1] `http://comments.gmane.org/gmane.comp.java.hadoop.hbase.devel/39017`

Part IV

Conclusion and Future Work

CHAPTER 10

CONCLUSION AND FUTURE WORK

Test-data generation is an important stage of software development. Manually generating test-data is a laborious and effort-consuming task. Many researchers (see Chapters 3) have proposed automated approaches to generate test data. Among the proposed approaches (Baars *et al.*, 2011; Fraser et Zeller, 2011; Fraser et Arcuri, 2012) have proven to be efficient in generating test data by using static analyses with SB-STDG. However, their analyses focus only on one or two UUT features, i.e., constants and conditional statements. Deeper static analyses on these two features and a focus on other UUT features, such as arguments, data members, methods, and relationships with other units, could improve SB-STDG. Hence, in this dissertation, our thesis was:

> Statically analyzing source code to identify and extract relevant information to exploit them in the SB-STDG process either directly or through CB-STDG offers more guidance and thus improves the efficiency and effectiveness of SB-STDG for object-oriented testing.

To prove our hypothesis, we proposed to consider many UUT features and analyzed them to derive information relevant to better guide CB-STDG.

10.1 CP Techniques to Analyze and Improve SB-STDG

10.1.1 CPA-STDG

We proposed CPA-STDG, which uses CP techniques to analyze and model the UUT. CPA-STDG models a program with its CFG by a CSP to efficiently explore execution paths, hence efficiently generating test data. We showed that modeling a program and its CFG by one CSP keeps the program structural semantics and therefore can helps to generate test data for any structural coverage criteria. The results showed the usability and effectiveness of CPA-STDG compared to the state of the art.

10.1.2 CSB-STDG

We proposed CSB-STDG, a new approach that combines CPA-STDG and SB-STDG to achieve high degree of code coverage. CSB-STDG simplifies the problem of test-data generation by solving it with CPA-STDG and using the solutions as a reduced search space

of SB-STDG. We identified two main steps, i.e., initial population and mutation operator, where CPA-STDG can be useful for SB-STDG. In each step, CPA-STDG feeds SB-STDG with relevant test-data candidates. We compared CSB-STDG with CPA-STDG and the state of the art of SB-STDG, i.e., eToc (Tonella, 2004). The results showed that CSB-STDG outperforms both techniques in terms of runtime and branch coverage. It could reach 89.5% branch coverage in less than 40 s, whereas eToc could not go beyond 85.78%, which was reached after 120 s. CPA-STDG performed badly: its best attained coverage is 78.94%.

10.1.3 CB-FF

We proposed f_{DC} and f_{DL}, two new fitness functions that analyze branch conditions and assign a hardness value for each branch using CP techniques, i.e., arity and constrainedness (Pesant, 2005). f_{DC} and f_{DL} prioritizes branches according to how hard they are to satisfy. To evaluate a test-datum candidate, f_{DC} and f_{DL} consider branches leading to the test target and use a hybrid evaluation. f_{DC} and f_{DL} dynamically compute a branch distance for the traversed branches and statically compute a branch distance for each non-executed branch, then all branch distances are adjusted by their hardness values and summed to determine the test-datum candidate fitness value. The results of an empirical study on a large number of benchmarks showed that evolutionary algorithm and simulated annealing with our new fitness functions are significantly more effective and efficient than with the largely used fitness functions from the literature, i.e., branch distance (Tracey *et al.*, 1998), approach level (Wegener *et al.*, 2001), and symbolic enhanced Baars *et al.* (2011).

10.2 Schema Theory to Analyze and Improve SB-STDG

We proposed EGAF-STDG, aSB-STDG approach that uses schema theory (Holland, 1975) to analyze and to tailor GA to better fit the test-data generation problem, thus reaching better coverage. We adapted the general form of schema theory for the test-data generation problem. EGAF-STDG identifies structures and properties associated with better performance by schema analysis, then incorporates them in the evolution phase with all fundamental GA operators (selection, crossover, mutation). The results of the comparison of a simple GA framework, i.e., (Baresel *et al.*, 2002; Wegener *et al.*, 2001; Harman et McMinn, 2010), for software test-data generation and EGAF-STDG showed that the latter outperforms the former. Thus, EGAF-STDG significantly reduced the number of evaluations needed to reach a given branch and achieved higher branch coverage than a simple GA framework.

10.3 Lightweight Static Analyses to Improve SB-STDG

We proposed IG-PR-IOOCC, a SB-STDG approach for unit-class testing. IG-PR-IOOCC relies on static analyses: (1) it statically analyzes the internal structure of a class-under-test (CUT) to reduce the search space and to define a seeding strategy and, (2) it statically analyzes the relationships between CUT and other classes to diversify class instances. IG-PR-IOOCC considers that the problem of test-data generation faces three difficulties: ($D1$) finding an adequate instance of the CUT and of each required class; ($D2$) finding an adequate sequence of method calls to put the instance of the CUT in a desired state; and, ($D3$) finding an adequate method to reach the test target. IG-PR-IOOCC reduces the search space of $D2$ and $D3$ by selecting sequences from a subset of methods that contains only methods relevant to the test target. A method is relevant to $D2$ iff it may change the state of an instance of the CUT and it is relevant for $D3$ iff it may reach the test target. To solve $D1$, we propose an instance generator based on a diversification strategy, instantiation of anonymous classes, and a seeding strategy to boost the search.

Results showed that IG-PR-IOOCC and its implementation JTExpert outperformed the state of the art, i.e., EvoSuite Fraser et Arcuri (2011, 2013c). JTExpert found a test-data suite and achieved a high code coverage (70%) in less than 10 s, whereas Evosuite reached 31 % coverage at 10 seconds and 61 % at 200 seconds. We showed on more than a hundred classes taken from different open-source Java libraries that JTExpert has a higher code coverage than EvoSuite and needs less time.

To evaluate the scalability of IG-PR-IOOCC and its implementation JTExpert on large software, we proposed an adaption of JTExpert to raise exceptions via branch coverage, i.e., an Exceptions-oriented Test-data Generation Approach. Then we applied it on a large open-source software the Apache Hadoop System. The results revealed 4,323 failures, two of them leading to JVM crashes.

10.4 Summary

This dissertation analyzed the benefits of using static analyses for SB-STDG. We identified six UUT features that influence the process of test-data generation: data members, methods, arguments, conditional statements, constants, and relationships with other units. We proposed four approaches that focus their static analyses on these six features: (1) focusing on arguments and conditional-statements of a UUT and using CP techniques, we defined CPA-STDG a CB-STDG approach and use it in the approach CSB-STDG as static analysis to guide SB-STDG in reducing its search space; (2) focusing on conditional statements and using CP techniques, we defined f_{DC} and f_{DL} two new fitness functions to guide SB-STDG;

(3) focusing on conditional-statements and using schema theory, we defined EGAF-STDG, a new SB-STDG framework; and, (4) focusing on arguments, conditional statements, constants, data members, methods, and relationship with other units, we defined IG-PR-IOOCC a new SB-STDG relying on static analysis techniques for object-oriented test-data generation. The results proved our thesis: statically analyzing UUT features improves SB-STDG. We found that analyzing more features provide more relevant information, hence better guidance for SB-STDG to reach a high degree of code coverage. Thus, all previously mentioned UUT features influence the process of test-data generation.

10.5 Limitations

Despite the above promising results, our thesis is threatened by the following limitations:

10.5.1 Limitations of CPA-STDG

CPA-STDG uses a code-to-code transformation, so it highly depends on the UUT source code. In some real world program the source code may be complex (e.g., complex data structures and part of the source code is a black box). In such cases, CPA-STDG alone could not be used to generate test data. We observed in Chapter 5 that CPA-STDG could not achieve a high code coverage. Because it uses the relaxation to simplify the source code, hence it could not translate all test-data generation problem into an equivalent CSP.

CPA-STDG uses some modeling constraints that translates a single method or procedure into an equivalent CSP. However, in OOP, a test-datum is a sequence of method calls. In such case, CPA-STDG requires a generator of sequences of method calls that also must define a test target to reach in each method of the sequence (e.g., a random sequences generator and a random branch to reach in each method).

10.5.2 Limitations of CSB-STDG

CSB-STDG uses relaxation to simplify a UUT, then it calls CPA-STDG to generate some potential test-data candidates. However, in some cases, an infeasible test-data generation problem may generate a feasible relaxed one. In such cases, CSB-STDG lacks a main advantage of a CB-STDG approach, which proves the infeasibility of a test-data generation problem. Therefore, CSB-STDG may not perform well in the presence of infeasible paths.

10.5.3 Limitations of CB-FF

CB-FF relies on symbolic-execution and relaxation. Even using relaxation to simplify complex expressions, symbolically analyzing the whole source code to evaluate all branches in

terms of arguments, data members, and global variables is a complex task; even more complex if a symbolic evaluation must be performed for every execution. The unavailability of an open-source symbolic-execution analyzer makes the implementation of CB-FF very difficult. We implemented a simple symbolic-execution analyzer that can analyze a small number of expressions and relax any other unsupported expression. Thus, to have better symbolic evaluation, we were obliged to perform our experiments on synthetic programs. Therefore, it is possible that, on real programs, CB-FF performs differently.

10.5.4 Limitations of EGAF-STDG

EGAF-STDG uses CB-FF to define its fitness function. Therefore, EGAF-STDG inherits the problem of symbolic evaluation of complex expressions. To validate EGAF-STDG, we used the same symbolic-execution analyzer and benchmarks as for evaluating CB-FF. Therefore, it is possible that, on real programs, EGAF-STDG performs differently.

10.5.5 Limitations of IG-PR-IOOCC

CPA-STDG, CSB-STDG, CB-FF, and EGAF-STDG use complex static analyses because they use CP techniques. Contrary to previous approaches, IG-PR-IOOCC uses lightweight static analyses instead of CP techniques. Hence, it is easy to implement and apply on real programs, but it lacks the main advantage of a CB-STDG approach, which is proving the infeasibility of a test-data generation problem. Therefore, IG-PR-IOOCC may not perform well in the presence of infeasible paths. Also, among all JTExpert components, the random search could be either a weak or advantageous link. Therefore, it is possible that with another search algorithm IG-PR-IOOCC performs differently.

The implementation of IG-PR-IOOCC, JTExpert, is unsafe and may lead to an undesirable system behavior during test-data generation process. To have better control over JTExpert, we performed experimentations using a user with very restrictive access to the file system. In such an environment JTExpert may not be allowed to execute some statements in the UUT. This may negatively influence the code coverage.

During the experimentation, we observed that JTExpert generates a set of test data that is difficult to follow and understand. Further, JTExpert does not generate an oracle for each test datum. These two observations raise a question that the test data generated by JTExpert, as it stands now, may not be reusable throughout a real software development cycle. For example, Figure 10.1 shows a test datum that was automatically generated to test class `org.joda.time.Minutes`. At first glance, this test datum seems difficult to follow and understand. Analyzing the first six comment lines, we can understand that test datum calls

```
1  /* Chromosome :
2  1)------>toStandardMinutes[]
3  2)------>parseMinutes[String],
4  3)------>plus[-606]
5  Covered Branches:[17, 1, 19, 2, 32, 20, 8, 30]*/
6  @Test public void TestCase3() throws Throwable {
7      long clsUTMinutesP1P0P2P0P0P1P0P0P0P1=87525275727380863L;
8      DateTimeZone clsUTMinutesP1P0P2P0P0P1P0P0P0P2=(DateTimeZone)DateTimeZone.UTC;
9      DateTime clsUTMinutesP1P0P2P0P0P1P0P0P0=new DateTime(clsUTMinutesP1P0P2P0P0P1P0P0P0P1,↩
           clsUTMinutesP1P0P2P0P0P1P0P0P0P2);
10     DateTime.Property clsUTMinutesP1P0P2P0P0P1P0P0=clsUTMinutesP1P0P2P0P0P1P0P0P0.millisOfDay();
11     long clsUTMinutesP1P0P2P0P0P1P0P1=81L;
12     AbstractInstant clsUTMinutesP1P0P2P0P0P1P0=clsUTMinutesP1P0P2P0P0P1P0P0.addToCopy(↩
           clsUTMinutesP1P0P2P0P0P1P0P1);
13     Date clsUTMinutesP1P0P2P0P0P1=clsUTMinutesP1P0P2P0P0P1P0.toDate();
14     LocalDateTime clsUTMinutesP1P0P2P0P0=LocalDateTime.fromDateFields(clsUTMinutesP1P0P2P0P0P1);
15     LocalTime clsUTMinutesP1P0P2P0=clsUTMinutesP1P0P2P0P0.toLocalTime();
16     TimeZone clsUTMinutesP1P0P2P1P1=TimeZone.getDefault();
17     DateTimeZone clsUTMinutesP1P0P2P1=DateTimeZone.forTimeZone(clsUTMinutesP1P0P2P1P1);
18     ReadableInstant clsUTMinutesP1P0P2=clsUTMinutesP1P0P2P0.toDateTimeToday(clsUTMinutesP1P0P2P1);
19     Period clsUTMinutesP1P0=new Period((ReadableInstant)null,clsUTMinutesP1P0P2);
20     Minutes clsUTMinutes=clsUTMinutesP1P0.toStandardMinutes();
21     String clsUTMinutesP2P1=new String("p");
22     Minutes clsUTMinutesP2=Minutes.parseMinutes(clsUTMinutesP2P1);
23     int clsUTMinutesP3P1=-606;
24     Minutes clsUTMinutesP3=clsUTMinutes.plus(clsUTMinutesP3P1);
25   }
```

Figure 10.1: A difficult to follow test datum that was automatically generated by JTExpert to test class `Minutes` in package `org.joda.time`

three main methods. It generates an instance of the UUT by calling the method `toStandardMinutes()` on an instance of class `org.joda.time.Period` and calls two methods on the instance of the UUT, i.e., `parseMinutes("p")` and `plus(-606)`. Further analyzing the test datum, we observe that the actual test starts at Line 20, ends at Line 24, and all the lines before generating an instance of `org.joda.time.Period`. Lines 7 to 19 are not at the heart of the test datum and they decrease both its readability and code quality. Also, this test datum cannot reveal an error in the implementation because, after the method calls at Lines 22 and 24, JTExpert did not insert assertions to check if the returned values are as expected.

10.6 Future Work

In this dissertation, we have verified our thesis and showed that statically analyzing internal UUT features improves SB-STDG.

Future work will be devoted to three research directions: (1) further experiments; (2) broadening and enlarging static analyses on UUT features; and, (3) statically analyzing new external sources to derive information about UUT that is relevant for test-cases generation. Below, we describe how we will plan to extend the work presented in this dissertation.

10.6.1 Further Experiments

We would like to extend the evaluations of our approaches, especially those involving CP techniques, i.e., CPA-STDG, CSB-STDG, CB-FF, and EGAF-STDG. First, we plan to enhance the implementation of CPA-STDG to cover some complex expressions (e.g., shifting operator). Second, IG-PR-IOOCC currently relies on a guided random search and supports branch coverage. We are working on adding further search algorithms, such as genetic algorithms and hill climbing. Also, we will follow three research directions: (1) enlarging the scope of JTExpert to support other structural testing criteria, such as data-flow coverage or mutation-coverage; (2) enhancing the instance generator to generate instances of classes that require understanding the context of their use; and, (3) studying the impact of seeding `null` with the instances of classes on the generation of null pointer exceptions. Finally, we would like to apply our approaches on industrial systems instead of being limited to synthetic and open-source programs.

10.6.2 Broadening and Enlarging Static Analyses on UUT Features

We observed that static analyses of UUT features improve the test-data generation process. This line of research remains promising, because it may offer means to deepen the static analyses on some features as well as to broaden them by adding some missing UUT features. In the future, we want to extend our static analyses on method members and local classes.

In our research work, we analyzed and proposed techniques to cover private and public methods, yet analyzing protected methods may improve further the code coverage. Protected methods, are not directly accessible. They are visible only from the enclosing class and its derived classes. In the absence of calls to a protected method there is no way to reach it except with creating a stub derived from the enclosing class that explicitly calls the protected method. We plan to use such a technique to cover any protected method.

We analyzed anonymous classes and proposed a technique to instantiate them and to improve their code coverage. We would like to generalize that technique to cover any local classes, hence further improving code coverage. We also plan to enlarge static analyses to cover class members, i.e., nested classes, in particular private and protected class members.

10.6.3 Statically Analyzing External Resources from UUT

This line of research aims to improve the code coverage as well as the readability and quality of generated test data.

In this research work, we stated some relationships between UUT and others units, i.e., subclasses, and showed the importance of this relationship in improving test-data generation.

In some cases the test-data generation problem may depend on relationships between UUT and external resources. In this case, a search that lacks this information and focuses only on internal UUT features may fail to generate a test datum. For example, let us suppose that the UUT takes as input a string that must be a path pointing to a configuration file. In such a case, a search that does not have that information may fail to generate a test datum.

In general, the source code does not contain all relevant information about a UUT and external resources such as specification documents or configuration files could be a precious source of relevant information about UUT and may help to improve the test data and test-cases generation process. Hence, in the future, we plan to identify external factors that may influence the test-cases generation process and develop static analyses focusing on them. External resources may also contain information relevant for oracle generation. Therefore, we would like to use external resources linked to a UUT to generate a relevant oracle for each test datum and improve the quality and readability of test cases.

REFERENCES

ADIR, A. et NAVEH, Y.(2011). Stimuli generation for functional hardware verification with constraint programming. P. van Hentenryck et M. Milano, éditeurs, *Hybrid Optimization*, Springer New York, vol. 45 de *Springer Optimization and Its Applications*. 509–558.

ALLEN, F. E.(1970). Control flow analysis. *SIGPLAN Not.*, 5, 1–19.

ALLEN, F. E. et COCKE, J.(1976). A program data flow analysis procedure. *Commun. ACM*, 19, 137–.

ALSHAHWAN, N. et HARMAN, M.(2011). Automated web application testing using search based software engineering. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 3–12.

ALSHRAIDEH, M. et BOTTACI, L.(2006). Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16, 175–203.

ANDREWS, J. H., HALDAR, S., LEI, Y. et LI, F. C. H.(2006). Tool support for randomized unit testing. *Proceedings of the 1st international workshop on Random testing*. ACM, 36–45.

ANDREWS, J. H., MENZIES, T. et LI, F. C.(2011). Genetic algorithms for randomized unit testing. *Software Engineering, IEEE Transactions on*, 37, 80–94.

ARCURI, A.(2010). It does matter how you normalise the branch distance in search based software testing. *ICST*. IEEE Computer Society, 205–214.

ARCURI, A. et BRIAND, L.(2014). A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24, 219–250.

ARCURI, A. et FRASER, G.(2011). On parameter tuning in search based software engineering. *Search Based Software Engineering*, Springer Berlin Heidelberg, vol. 6956 de *Lecture Notes in Computer Science*. 33–47.

ARCURI, A. et YAO, X.(2008). Search based software testing of object-oriented containers. *Information Sciences*, 178, 3075–3095.

BAARS, A. I., HARMAN, M., HASSOUN, Y., LAKHOTIA, K., MCMINN, P., TONELLA, P. et VOS, T. E. J.(2011). Symbolic search-based testing. P. Alexander, C. S. Păsăreanu et J. G. Hosking, éditeurs, *ASE*. IEEE, 53–62.

BAKER, J. E.(1987). Reducing bias and inefficienry in the selection algorithm. J. J. Grefenstette, éditeur, *ICGA*. Lawrence Erlbaum Associates, 14–21.

BARDIN, S., BOTELLA, B., DADEAU, F., CHARRETEUR, F., GOTLIEB, A., MARRE, B., MICHEL, C., RUEHER, M. et WILLIAMS, N.(2009). Constraint-based software testing. *Journée du GDR-GPL*, 9, 1.

BARDIN, S. et HERRMANN, P.(2008). Structural testing of executables. *ICST*. 22–31.

BARESEL, A., STHAMER, H. et SCHMIDT, M.(2002). Fitness function design to improve evolutionary structural testing. *Proceedings of the Genetic and Evolutionary Computation Conference*. 1329–1336.

BARRETT, C. et TINELLI, C.(2007). CVC3. W. Damm et H. Hermanns, éditeurs, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*. Springer-Verlag, vol. 4590 de *Lecture Notes in Computer Science*, 298–302. Berlin, Germany.

BASILI, V. R. et WEISS, D. M.(1984). A methodology for collecting valid software engineering data. *Software Engineering, IEEE Transactions on*, SE-10, 728 –738.

BHATTACHARYA, N., SAKTI, A., ANTONIOL, G., GUÉHÉNEUC, Y.-G. et PESANT, G.(2011). Divide-by-zero exception raising via branch coverage. M. B. Cohen et M. Ó. Cinnéide, éditeurs, *SSBSE*. Springer, vol. 6956 de *Lecture Notes in Computer Science*, 204–218.

BINKLEY, D. et HARMAN, M.(2004). Analysis and visualization of predicate dependence on formal parameters and global variables. *Software Engineering, IEEE Transactions on*, 30, 715–735.

BLICKLE, T. et THIELE, L.(1996). A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4, 361–394.

BOTELLA, B., DELAHAYE, M., KOSMATOV, N., P. MOUY, M. R. et WILLIAMS, N.(2009). Automating structural testing of c programs: Experience with pathcrawler. *Fourth International Workshop on the Automation of Software Test*. 70–78.

BOTELLA, B., GOTLIEB, A., MICHEL, C., RUEHER, M. et TAILLIBERT, P.(2002). Utilisation des contraintes pour la génération automatique de cas de test structurels. *Technique et sciences informatiques*, 21, 21–45.

BOYER, R. S., ELSPAS, B. et LEVITT, K. N.(1975). Select–a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10, 234–245.

BRANDIS, M. et MÖSSENBÖCK, H.(1994). Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 14, 1684–1698.

CHOCO(2012). Choco is an open source Java constraint programming library. `http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf`. [Online; accessed 10-Fab-2012].

CIUPA, I., LEITNER, A., ORIOL, M. et MEYER, B.(2008). Artoo. *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 71–80.

CLARKE, L.(1976). A system to generate test data and symbolically execute programs. *Software Engineering, IEEE Transactions on*, SE-2, 215 – 222.

COLLAVIZZA, H. et RUEHER, M.(2006). Exploration of the capabilities of constraint programming for software verification. *Tools and Algorithms for the Construction and Analysis of Systems*. vol. 3920, 0302–9743.

COLLAVIZZA, H., RUEHER, M. et HENTENRYCK, P. V.(2010). Cpbpv: a constraint-programming framework for bounded program verification. *Constraints*, 15, 238–264.

CSALLNER, C. et SMARAGDAKIS, Y.(2004). Jcrasher: an automatic robustness tester for java. *Software: Practice and Experience*, 34, 1025–1050.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N. et ZADECK, F. K.(1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13, 451–490.

DEMILLO, R. A. et OFFUTT, A. J.(1993). Experimental results from an automatic test case generator. *ACM Trans. Softw. Eng. Methodol.*, 2, 109–127.

DEVELOPMENT TOOLS (JDT), E. J.(2013). The jdt project provides the tool plug-ins that implement a java ide supporting the development of any java application, including eclipse plug-ins. [Online; accessed 02-FEB-2013].

FERRANTE, J., OTTENSTEIN, K. J. et WARREN, J. D.(1987). The program dependence graph and its use in optimization. *Transactions on Programming Languages and Systems*, 9, 319–349.

FORD, B.(2007). One Source Shortest Path: Dijkstra's Algorithm. `http://compprog.wordpress.com/2007/12/01/one-source-shortest-path-dijkstras-algorithm/`. [Online; accessed 30-Oct-2011].

FOSTER, K. A.(1984). Sensitive test data for logic expressions. *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, 9, 120–125.

FRASER, G. et ARCURI, A.(2011). Evosuite: automatic test suite generation for object-oriented software. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 416–419.

FRASER, G. et ARCURI, A.(2012). The seed is strong: Seeding strategies in search-based software testing. *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 121–130.

FRASER, G. et ARCURI, A.(2013a). 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *ESE Journal*, 1–29.

FRASER, G. et ARCURI, A.(2013b). Evosuite at the sbst 2013 tool competition. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0, 406–409.

FRASER, G. et ARCURI, A.(2013c). Whole test suite generation. *IEEE Transactions on Software Engineering*, 39, 276 –291.

FRASER, G. et ZELLER, A.(2011). Exploiting common object usage in test case generation. *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 80–89.

FRASER, G. et ZELLER, A.(2012). Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38, 278–292.

GARFINKEL, S.(2005). History's Worst Software Bugs. `http://www.wired.com/print/software/coolapps/news/2005/11/69355`. [Online; accessed 23-Mai-2010].

GODEFROID, P., KLARLUND, N. et SEN, K.(2005). Dart: directed automated random testing. *SIGPLAN Not.*, 40, 213–223.

GOLDBERG, D. E.(1989). *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley.

GOLDBERG, D. E. et DEB, K.(1991). A comparative analysis of selection schemes used in genetic algorithms. *Urbana*, 51, 61801–2996.

GOODENOUGH, J. B. et GERHART, S. L.(1975). Toward a theory of test data selection. *Software Engineering, IEEE Transactions on*, 156–173.

GOTLIEB, A.(2009). Euclide: A constraint-based testing framework for critical c programs. *ICST*. IEEE Computer Society, 151–160.

GOTLIEB, A., BOTELLA, B. et RUEHER, M.(2000). A clp framework for computing structural test data. *Computational Logic*, 1861, 399–413.

GUPTA, A.(2008). From hardware verification to software verification: Re-use and relearn. K. Yorav, éditeur, *Hardware and Software: Verification and Testing*, Springer Berlin / Heidelberg, vol. 4899 de *Lecture Notes in Computer Science*. 14–15.

HARMAN, M., FOX, C., HIERONS, R., HU, L., DANICIC, S. et WEGENER, J.(2002). Vada: a transformation-based system for variable dependence analysis. *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*. 55–64.

HARMAN, M., HASSOUN, Y., LAKHOTIA, K., MCMINN, P. et WEGENER, J.(2007). The impact of input domain reduction on search-based test data generation. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, New York, NY, USA, ESEC-FSE '07, 155–164.

HARMAN, M. et MCMINN, P.(2010). A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Software Engineering, IEEE Transactions on*, 36, 226–247.

HOFFMANN, M. R., JANICZAK, B. et MANDRIKOV, E.(2012). Jacoco is a free code coverage library for java. [Online; accessed 01-OCT-2013].

HOLLAND, J. H.(1975). Adaptation in natural and artificial systems, university of michigan press. *Ann Arbor, MI*, 1, 5.

IASOLVER(2012). IAsolver is an Interval Arithmetic Constraint Solver. `http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html`. [Online; accessed 10-Fab-2013].

IBM(2009a). Ibm ilog cplex v12.1 user's manual. *IBM ILOG*.

IBM(2009b). Ibm ilog solver v6.7 user's manual. *IBM ILOG*.

INCE, D. C.(1987). The automatic generation of test data. *The Computer Journal*, 30, 63–69.

INKUMSAH, K. et XIE, T.(2007). Evacon: A framework for integrating evolutionary and concolic testing for object-oriented programs. *Proc. 22nd IEEE/ACM ASE*. 425–428.

JAFFAR, J. et MAHER, M. J.(1994). Constraint logic programming - a survey. *Logic Programming*, 20, 503–581.

JONES, T.(1994). A description of hollandś royal road function. *Evol. Comput.*, 2, 409–415.

JUNIT(2013). Junit is a simple framework to write repeatable tests. `http://www.junit.org`. [Online; accessed 19-JUN-2013].

KHURSHID, S., PĂSĂREANU, C. S. et VISSER, W.(2003). Generalized symbolic execution for model checking and testing. *Tools and Algorithms for the Construction and Analysis of Systems*, Springer. 553–568.

KING, J. C.(1976). Symbolic execution and program testing. *Commun. ACM*, 19, 385–394.

KONG, J. et YAN, H.(2010). Comparisons and analyses between rtca do-178b and gjb5000a, and integration of software process control. *International Conference on Advanced Computer Theory and Engineering*. vol. 6, 367–372.

KOREL, B.(1990). Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16, 870–879.

LAKHOTIA, K., TILLMANN, N., HARMAN, M. et DE HALLEUX, J.(2010). Flopsy - search-based floating point constraint solving for symbolic execution. A. Petrenko, A. Simao et J. Maldonado, éditeurs, *Testing Software and Systems*, Springer Berlin / Heidelberg, vol. 6435 de *Lecture Notes in Computer Science*. 142–157.

LARSON, E. et AUSTIN, T.(2003). High coverage detection of input-related security faculties. *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. USENIX Association, Berkeley, CA, USA, 9–9.

LUKASIEWYCZ, M., GLAß, M., REIMANN, F. et TEICH, J.(2011). Opt4j: a modular framework for meta-heuristic optimization. N. Krasnogor et P. L. Lanzi, éditeurs, *GECCO*. ACM, 1723–1730.

LUKE, S., PANAIT, L., BALAN, G., PAUS, S., SKOLICKI, Z., BASSETT, J., HUBLEY, R. et CHIRCOP, A.(2010). Ecj: A java-based evolutionary computation research system.

MACHADO, P., VINCENZI, A. et MALDONADO, J. C.(2010). Software testing: An overview. *Testing Techniques in Software Engineering*, Springer. 1–17.

MALBURG, J. et FRASER, G.(2011). Combining search-based and constraint-based testing. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. IEEE Computer Society, ACM, New York, NY, USA, ISSTA '11, 436–439.

MANN, H. B. et WHITNEY, D. R.(1947). On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, 50–60.

MCCABE, T. J.(1976). A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2, 308–320.

MCMINN, P.(2004). Search-based software test data generation: a survey. *Software Testing Verification & Reliability*, 14, 105–156.

MCMINN, P., HARMAN, M., BINKLEY, D. et TONELLA, P.(2006). The species per path approach to searchbased test data generation. *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, New York, NY, USA, ISSTA '06, 13–24.

MCMINN, P., HARMAN, M., LAKHOTIA, K., HASSOUN, Y. et WEGENER, J.(2012a). Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. *Software Engineering, IEEE Transactions on*, 38, 453–477.

MCMINN, P., SHAHBAZ, M. et STEVENSON, M.(2012b). Search-based test input generation for string data types using the results of web queries. *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. IEEE, 141–150.

MCMINN, P., STEVENSON, M. et HARMAN, M.(2010). Reducing qualitative human oracle costs associated with automatically generated test data. *Proceedings of the First International Workshop on Software Test Output Validation*. ACM, 1–4.

MICHAEL, C., MCGRAW, G. et SCHATZ, M.(2001). Generating software test data by evolution. *Software Engineering, IEEE Transactions on*, 27, 1085–1110.

MILLER, W. et SPOONER, D.(1976). Automatic generation of floating-point test data. *Software Engineering, IEEE Transactions on*, SE-2, 223 – 226.

MITCHELL, M.(1999). An introduction to genetic algorithms. *Cambridge, Massachusetts London, England, Fifth printing*.

MITCHELL, M., FORREST, S. et HOLLAND, J. H.(1992). The royal road for genetic algorithms: Fitness landscapes and ga performance. *Proceedings of the first european conference on artificial life*. Cambridge: The MIT Press, 245–254.

MOSS, A.(2008). Constraint patterns and search procedures for cp-based random test generation. *Proceedings of the 3rd international Haifa verification conference on Hardware and software: verification and testing*. Springer-Verlag, Berlin, Heidelberg, HVC'07, 86–103.

MOSS, A.(2010). Constraint programming with arbitrarily large integer variables. A. Lodi, M. Milano et P. Toth, éditeurs, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Springer Berlin / Heidelberg, vol. 6140 de *Lecture Notes in Computer Science*. 252–266.

MÜHLENBEIN, H. et SCHLIERKAMP-VOOSEN, D.(1993). Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evolutionary computation*, 1, 25–49.

MYERS, G. J.(1979). *The art of software testing*. John Wiley and Sons.

NAVEH, Y., RIMON, M., JAEGER, I., KATZ, Y., VINOV, M., MARCUS, E. et SHUREK, G.(2007). Constraint-based random stimuli generation for hardware verification. *The AI magazine*, American Association for Artificial Intelligence, vol. 28. 13–30.

ORIAT, C.(2005). Jartege: a tool for random generation of unit tests for java classes. *Quality of Software Architectures and Software Quality*, Springer. 242–256.

PACHECO, C. et ERNST, M. D.(2005). *Eclat: Automatic generation and classification of test inputs*. Springer.

PACHECO, C., LAHIRI, S. K., ERNST, M. D. et BALL, T.(2007). Feedback-directed random test generation. *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 75–84.

PARASOFT(2013). Jtest: Java static analysis, code review, unit testing, security. [Online; accessed 13-OCT-2013].

PARGAS, R. P., HARROLD, M. J. et PECK, R. R.(1999). Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9, 263–282.

PESANT, G.(2005). Counting solutions of csps: A structural approach. L. P. Kaelbling et A. Saffiotti, éditeurs, *IJCAI*. Professional Book Center, 260–265.

POHLHEIM, H.(2006). GEATbx: Genetic and Evolutionary Algorithm Toolbox for use with MATLAB Documentation. `http://www.geatbx.com/docu/index.html`. [Online; accessed 10-Feb-2013].

PRESSMAN, R. S.(1992). *Software Engineering: A Practitioner's Approach 3rd edition*. McGraw-Hill.

PĂSĂREANU, C. S. et RUNGTA, N.(2010). Symbolic pathfinder: symbolic execution of java bytecode. C. Pecheur, J. Andrews et E. D. Nitto, éditeurs, *ASE*. ACM, 179–180.

PĂSĂREANU, C. S., RUNGTA, N. et VISSER, W.(2011). Symbolic execution with mixed concrete-symbolic solving. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, ISSTA '11, 34–44.

PUGET, J.-F.(1995). Applications of constraint programming. U. Montanari et F. Rossi, éditeurs, *Principles and Practice of Constraint Programming, CP '95*, Springer Berlin / Heidelberg, vol. 976 de *Lecture Notes in Computer Science*. 647–650.

RAPPS, S. et WEYUKER, E.(1985). Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11, 367–375.

REFLECTION(2013). The java reflection api. `http://docs.oracle.com/javase/tutorial/reflect/`. [Online; accessed 01-SEP-2013].

REFLECTIONS(2013). Java runtime metadata analysis. `https://code.google.com/p/reflections/`. [Online; accessed 01-SEP-2013].

RÉGIN, J.-C.(2011). Global constraints: a survey. *Hybrid optimization*, Springer. 63–134.

RIBEIRO, J. C. B., ZENHA-RELA, M. A. et DE VEGA, F. F.(2008). Strongly-typed genetic programming and purity analysis: input domain reduction for evolutionary testing problems. *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, 1783–1784.

RTCA(1992). Software consideration in airborne systems and equipment certification. *Software verification process*.

SAKTI, A., GUÉHÉNEUC, Y. et PESANT, G.(2013). Constraint-based fitness function for search-based software testing. C. P. Gomes et M. Sellmann, éditeurs, *Integration of AI*

and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings. Springer, vol. 7874 de *Lecture Notes in Computer Science*, 378–385.

SAKTI, A., GUÉHÉNEUC, Y.-G. et PESANT, G.(2011). Cp-sst : approche basée sur la programmation par contraintes pour le test structurel du logiciel. *Septitièmes Journées Francophones de Programmation par Contraintes (JFPC)*. 289–298.

SAKTI, A., GUÉHÉNEUC, Y.-G. et PESANT, G.(2012). Boosting search based testing by using constraint based testing. G. Fraser et J. T. de Souza, éditeurs, *SSBSE*. Springer, vol. 7515 de *Lecture Notes in Computer Science*, 213–227.

SAKTI, A., PESANT, G. et GUÉHÉNEUC, Y.-G.(2014). Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, PP, 1–1.

SEN, K. et AGHA, G.(2006). Cute and jcute: concolic unit testing and explicit path model-checking tools. *Proceedings of the 18th international conference on CAV*. Springer-Verlag, Berlin, Heidelberg, CAV'06, 419–423.

SEN, K., MARINOV, D. et AGHA, G.(2005). Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30, 263–272.

STAATS, M. et PĂSĂREANU, C.(2010). Parallel symbolic execution for structural test generation. *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, New York, NY, USA, ISSTA '10, 183–194.

TAHCHIEV, P., LEME, F., MASSOL, V. et GREGORY, G.(2010). *JUnit in action*. Manning Publications Co.

TASSEY, G.(2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 02–3.

TEAM, R. C. *ET AL.*(2012). R: A language and environment for statistical computing. [Online; accessed MAY-2014].

TILLMANN, N. et DE HALLEUX, J.(2008). Pex-white box test generation for .net. B. Beckert et R. Hahnle, éditeurs, *Tests and Proofs*, Springer Berlin / Heidelberg, vol. 4966 de *Lecture Notes in Computer Science*. 134–153.

TONELLA, P.(2004). Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29, 119–128.

TRACEY, N., CLARK, J., MANDER, K. et MCDERMID, J.(2000). Automated test-data generation for exception conditions. *Software-Practice and Experience*, 30, 61–79.

TRACEY, N., CLARK, J. A., MANDER, K. et MCDERMID, J. A.(1998). An automated framework for structural test-data generation. *ASE*. 285–288.

VARGHA, A. et DELANEY, H. D.(2000). A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25, 101–132.

VINCENZI, A., DELAMARO, M., HÖHN, E. et MALDONADO, J. C.(2010). Functional, control and data flow, and mutation testing: Theory and practice. *Testing Techniques in Software Engineering*, Springer. 18–58.

VISSER, W., PĂSĂREANU, C. S. et PELÁNEK, R.(2006). Test input generation for java containers using state matching. *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 37–48.

VISSER, W., PĂSĂREANU, C. S. et KHURSHID, S.(2004). Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29, 97–107.

VOS, T.(2013). Sbst contest: Java unit testing at the class level.

WAPPLER, S. et WEGENER, J.(2006). Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. *Proceedings of the 8th annual conference on Genetic and evolutionary computation*. ACM, 1925–1932.

WATSON, R. A., HORNBY, G. S. et POLLACK, J. B.(1998). Modeling building-block interdependency. *Parallel Problem Solving from Nature-PPSN V*, Springer Berlin Heidelberg, vol. 1498 de *Lecture Notes in Computer Science*. 97–106.

WEGENER, J., BARESEL, A. et STHAMER, H.(2001). Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43, 841 – 854.

WEISSTEIN, E. W.(2014). "Bonferroni Correction." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/BonferroniCorrection.html`. [Online; accessed December-2014].

WHITLEY, L. D.(1989). The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. J. D. Schaffer, éditeur, *ICGA*. Morgan Kaufmann, 116–123.

WILLIAMS, N., MARRE, B., MOUY, P. et ROGER, M.(2005). Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. *European Dependable Computing Conference*. vol. 3463, 281–292.

XIE, T., MARINOV, D., SCHULTE, W. et NOTKIN, D.(2005). Symstra: A framework for generating object-oriented unit tests using symbolic execution. *Tools and Algorithms for the Construction and Analysis of Systems*, Springer. 365–381.

XIE, T., TILLMANN, N., DE HALLEUX, J. et SCHULTE, W.(2009). Fitness-guided path exploration in dynamic symbolic execution. *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*. 359 –368.

YIN, R. K.(2014). *Case study research: Design and methods*. Sage publications.

Z3(2012). An efficient theorem prover. `http://research.microsoft.com/projects/Z3`. [Online; accessed 15-Jan-2012].

ZHANG, S., SAFF, D., BU, Y. et ERNST, M. D.(2011). Combined static and dynamic automated test generation. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, ISSTA '11, 353–363.